AEROSPACE REPORT NO.
ATR-93(3778)-5

Specification and Correctness Proofs of the MSX Ada Software

30 September 1993

Prepared by

T. K. MENAS, J. M. BOULER, AND J. E. DONER
Trusted Computer Systems Department
Computer Science and Technology Subdivision
Computer Systems Division
Engineering and Technology Group

Prepared for

DEPARTMENT OF DEFENSE
Ft. George G. Meade, MD 20744-6000

Engineering and Technology Group

THE AEROSPACE CORPORATION
El Segundo, California

AEROSPACE REPORT NO.
ATR-93(3778)-5

SPECIFICATION AND CORRECTNESS PROOFS OF THE MSX ADA SOFTWARE

Prepared by

T. K. MENAS, J. M. BOULER, AND J. E. DONER
Trusted Computer Systems Department
Computer Science and Technology Subdivision
Computer Systems Division
Engineering and Technology Group

30 September 1993

Engineering and Technology Group
THE AEROSPACE CORPORATION
El Segundo, CA 90245-4691

Prepared for

DEPARTMENT OF DEFENSE
Ft. George G. Meade, MD 20744-6000

Report No.
ATR-93(3778)-5

SPECIFICATION AND CORRECTNESS PROOFS OF THE MSX ADA SOFTWARE

Prepared

[illegible]
T. K. Menas

[illegible]
J. M. Bouler

John E. Doner
J. E. Doner

Beth Levy
B. H. Levy, Principal Investigator
Computer Assurance Section

Approved

[illegible]
D. B. Baker, Director
Trusted Computer Systems Department

Carl Sunshine
C. A. Sunshine, Principal Director
Computer Science and Technology Subdivision

cc: R2 (less encl)
R2SPO, Darnauer (less encl)

M/R: I received the report "Specifications and Correctness Proofs for Portions of the MSX Ada Software" from T. Menas, J. Bouler, and J. Doner, authors and SDVS researchers at The Aerospace Corporation. This document is to be given open and public distribution as an Aerospace Technical Report, ATR-93(3778)-5. I have examined this document and, in my opinion as COR for this unclassified research, the report is also unclassified and will not be detrimental to NSA's interest if released as requested. This report will also be reviewed and approved/disapproved for publication release by the MSX Pentagon's point-of-contact, LCOL Bruce Guilmain.

Dorothy L Darnauer 3-22-94
DOROTHY L. DARNAUER, R2SPO, 961-1264s, Mar 22, 94, ccj

CONCUR:

C. Terrence Ireland, R2	C. Terrence Ireland	dtd 22 Mar 94
Brian D. Snow, R2/TD	Brian D. Snow	dtd 22 MAR 94
Cathy Johnson, A/TCAO	Cathy Johnson	dtd 23 Mar 94
Michael B. Saft, TTTRO	[illegible]	dtd 28 Mar 94
Helen G. Cantor, Chief, I11	[illegible]	dtd 4 April 94
Bruce Guilmain, LCOL (USAF), BMDO/DTS	Bruce D Guilmain	dtd 10 MAY 94

Abstract

The Midcourse Space Experiment (MSX) is a Strategic Defense Initiative Organization program whose primary purpose is to conduct tracking event experiments of targets/phenomena in midcourse.

In this report we give a detailed account of the SDVS verification of a modified portion of the MSX tracking processor software. We present a functional overview of this part of the software, discuss and fully annotate our modifications to it, list the SDVS enhancements implemented during the course of the project, and present a proof of correctness of a simple but nontrivial specification.

Contents

Acknowledgments

We are especially indebted to Richard Waddell and Shane Hutton of the Johns Hopkins University Applied Physics Laboratory for their help in discussions of issues raised in this report. We are also grateful for the help given to us by Mark Bouler, Ivan Filippenko, Beth Levy, David Martin, and Leo Marcus of The Aerospace Corporation.

1 Introduction

The purpose of this report is to document a 1993 joint project between The Aerospace Corporation (Aerospace) and the Johns Hopkins University Applied Physics Laboratory (JHU/APL) to verify a portion of the Midcourse Space Experiment (MSX) spacecraft tracking processor software (the target software) using SDVS. The project consisted of three major parts: (1) enhancements to SDVS, primarily to the SDVS Ada translator and to the SDVS proof language; (2) modifications to the target software required by the SDVS Ada translator, and (3) proofs of and specifications for the modified target software.

Although we will give a very brief overview of the MSX software we are attempting to verify, we assume that the reader is acquainted with [1]. Detailed accounts of the tracking processor design and software are given in [2] and [3]. The target software builds and processes application-level messages from serial digital commands relayed from the command processor to the tracking processor. There are eleven types of application-level messages. One of these, the data-structure memory-load application message (data-structure message, for short), can modify up to 109 tracking parameters. At the present time only 61 of these 109 parameters are specified; the others are for future use. The number of commands that are required to form a data-structure message is a function of the type of tracking parameter that the data-structure message modifies. Once built, the data-structure message is stored in EEPROM, RAM, or both. Our goal was to verify that the data-structure messages are built according to their specification. The specifications for data-structure messages are described in [4].

The MSX software is written in Ada and in 1750A assembly.[1] The heart of the target software consists of the three Ada tasks – BUILD, PROCESS_MSG, and MANAGE_MSG_RETRIEVAL – in the package APP_MSGS, and the interrupt-driven Ada procedure CMD_IN_HANDLER in the package SERIAL_DIG_CMDS. When an interrupt occurs signaling that a command is ready to be retrieved from a designated serial port, CMD_IN_HANDLER services the interrupt by retrieving the command and storing it in a command buffer. For an infinite number of times, if the command buffer is not empty, BUILD retrieves the command buffer, constructs application-level messages from these commands, places them in a circular message queue, and waits for a rendezvous with MANAGE_MSG_RETRIEVAL. Also for an infinite number of times, if the message queue is not empty, PROCESS_MSG will rendezvous with MANAGE_MSG_RETRIEVAL to retrieve and process a message from the message queue. It stores data-structure messages in EEPROM, RAM, or both. As its name indicates, MANAGE_MSG_RETRIEVAL synchronizes the other two tasks.

The three tasks and the interrupt-driven procedure are essentially four tasks. It is important to note two facts: (1) the target software does not include a main program (or scheduler) to execute the tasks and the interrupt-driven procedure, and (2) SDVS does not currently handle Ada tasking. As a consequence, we translated the tasks into procedures and embedded these procedures in a program specifying an order of their execution. However, the translation of the tasks into procedures altered an important aspect of their executions;

[1]We are focusing on the software written in Ada. SDVS has the capability to verify programs written in a subset of 1750A assembly language as well, but this was not done in the current effort.

they could no longer be interleaved. Their execution as procedures, which is determined by the program, is one of the many possible execution sequences for the tasks. But the specification of the end result, namely that an infinite sequence of commands be processed into a corresponding infinite sequence of messages, remained the same for both the original and the modified MSX software.

Thus, Aerospace wrote the scheduler for the procedures that were tasks in the original software. Furthermore, we had to alter these procedures in various ways; these changes are explained in Section 4 and documented (statement by statement) in the actual modified target software listed in Appendix A. Some of the changes are equivalent to the original MSX code and some are not. We did not verify the correctness of the original code. However, we verified certain properties of the modified code that should also describe the original code; if the modified code did not satisfy the properties, then the original code would not have satisfied the properties. As a pragmatic consideration, if the modified code had not satisfied the properties, then, using the specifications, we could have constructed the appropriate test cases to demonstrate that the original code is incorrect. There is a range of possibilities in applying formal methods: e.g. formal specification and no proof, specification with hand-proof, specification with automated proof, etc. The MSX application was intended to stress our current SDVS capabilities by applying formal methods where possible, and it has been very useful for this. We worked with about 1000 lines of Ada code and tried to keep the modified version as close as possible to the original. The modified version of the code builds and processes messages, but without concurrent processes and a few other Ada constructs.

Section 2 presents a chronology of the MSX verification project using SDVS. After selecting the target software, we "walked through" the code to study it and itemize the Ada constructs not handled by the SDVS Ada translator. We also analyzed the parts of the code that interface with 1750A assembly language and the "intrinsic" functions supplied by the Tartan compiler, which is used for the compilation of the MSX software. The walk-through and the correspondence between Aerospace and JHU/APL were instrumental in the discovery of two errors in the target software.[2] After this initial phase of the project, we began the task of verification. Our approach was to provide proofs of correctness for increasingly complex situations. For our purposes, a correctness proof involves a program, a condition on its input (the input condition), and a specification relating the input to the output (the output condition). Complexity arose primarily from the generality of the scheduler and the type of input allowed by the input condition.

Section 3 discusses the enhancements we made to SDVS for the verification project, and Section 4 discusses modifications we made to the target software.

Section 5 presents some of the details of the proofs. The proofs appear in Appendix B. The scheduler that we wrote for CMD_IN_HANDLER, BUILD and PROCESS_MSG limits drastically the executions it encompasses; that is, the situations that arise in these executions, although possible, are really quite simple. For example, in every one of the allowed executions, the message queue is either empty or contains exactly one message, and the command buffer cannot be already full when another command is to be buffered. Thus, the message queue

[2]We note that at the time the errors were discovered, JHU/APL had not tested the code extensively.

can never overflow, and commands can never be lost. If there are any errors in the target software, one would expect them to be in precisely these boundary situations. It is really quite simple to write a scheduler that allows these boundary conditions to occur. We were forced to write this simple scheduler, because a proof of correctness for a more general scheduler would have been too difficult to construct in the time allotted for the one-year verification project. However, we think it is possible in SDVS to prove correctness for more general schedulers.

In Section 6 we discuss and present results from an approach involving the offline characterization capability of SDVS, which we think would facilitate proofs of correctness for more general schedulers. An approach to the verification effort using offline characterization would also reduce the size of proofs and the storage required for them, probably allowing the user to prove the correctness of the four large messages that we were unable to prove because of storage and time considerations.[3]

Offline characterization allows one to prove in SDVS an *adalemma* for a subprogram of a main program characterizing the results of the execution of the subprogram in a specific environment. It is, in fact, a proof of correctness for the subprogram. Instead of symbolically executing a subprogram when it is called in the SDVS execution of the main program, one may apply the adalemma for the subprogram and bypass its execution. This approach saves time and storage in a proof of correctness of a main program in which there are repeated calls to the subprogram. It also has the advantage of modularizing long proofs.

Section 7 contains our conclusions on the 1993 MSX verification project using SDVS.

[3]This verification project ended in fiscal year 1993.

2 Chronology of the MSX Verification Experiment

In this section we highlight the chronology of the 1993 MSX verification project using SDVS. We first note that many of the tasks constituting the project occurred throughout fiscal year 1993 and and some tasks were done concurrently. For example, the enhancements to SDVS necessitated by the project continued throughout its duration, as was the fixing of SDVS bugs not previously detected.

We first studied the documentation for the tracking processor software (reports [2], [3] and [4]) and then looked at the actual code. In our walkthrough we examined 18 library units that fell either in the area or on the periphery of our target software and noted the following:

(i) the library units and, specifically, the parts (targets) thereof needed in the building or processing of data-structure messages;

(ii) the Ada constructs in the targets not handled by the SDVS Ada translator;

(iii) the targets whose bodies were written in 1750A assembly code; and

(iv) the nonstandard Ada function targets provided by the Tartan compiler (some of the "intrinsic" functions).

The part of the MSX code we selected for verification was chosen partially because of its importance to the MSX experiment as a whole. In the process of selecting the targets, we had to select, for our purposes, two demarcation points in the software: the point at which the input stream of commands begins and the point at which the writing of data-structure messages to EEPROM ends. The first selection was easy: the input stream of commands begins at the servicing of an interrupt by CMD_IN_HANDLER.

The second selection was only slightly more difficult. PROCESS_MSG processes all application-level messages. If the message is a data-structure message and is to be written in EEPROM, it calls EEPROM_DATA.STORE_DATA_STRUC_IN_EEPROM.
Although STORE_DATA_STRUC_IN_EEPROM has an Ada body, this body repeatedly calls subprograms written in 1750A assembly. For this reason, we decided to consider that the messages have been written to EEPROM at the call to EEPROM_DATA.STORE_DATA_STRUC_IN_EEPROM.

The walkthrough determined our real target software: the targets within the two points of demarcation and any other part of the Ada code they required.

We partitioned the Ada constructs of item (ii) above into two groups: (1) those we would add to the SDVS Ada translator, and (2) those we would replace by equivalent Ada constructs (for example, hexadecimal literals by decimal literals) or simply replace by nonequivalent constructs (for example, tasks by procedures). We then started to implement the constructs in the first group into the SDVS Ada translator and made the appropriate modifications to the code for the constructs in the second group.[4]

[4]These are described in Sections 3 and 4 of this report.

In Section 4 we discuss the targets written in 1750A assembly and our handling of them. In a few cases, we wrote simple Ada code for their bodies that altered their functionality. For example, our version of the CHECK_PARITY function simply returns the boolean value *true*, whereas the original function returns either *true* or *false*, depending on the parity of the object to which it is applied. This change in the function alters the behavior of the main program. However it does not alter the behavior of the main program under the assumptions that all inputs have good parity, which is, in fact, one of our assumptions.

Common instances of calls to routines written in 1750A assembly language were calls to MEMORY_MANAGER.CONV_TYPE to convert from one type to another. We substituted these calls by explicit Ada type conversions: JHU/APL did not use these type conversions, because they were not handled properly by the Tartan compiler.

The actual MSX software is compiled at JHU/APL by the Tartan compiler. To test the target code using the Verdix Ada compiler available at Aerospace, we wrote Ada programs for the target "intrinsic" functions provided by the Tartan compiler: AND_I, AND_WORD, AND_BYTE, and OR_I. In the final SDVS correctness proofs, we characterized these functions by adalemmas. We used the Verdix compiler to test our modifications for syntactic correctness at every step of the modification process. In fact, we also tested several versions of the original code by executing it with various inputs. These versions were not identical to the original code because we replaced the routines that were written in 1750A assembly language, and we provided bodies for the Tartan-supplied "intrinsic" functions. But, we were able to test versions that were close to the original; for example, we were able to test a version with the original tasks.

Once we made the modifications and added the capability to translate subtypes to the SDVS Ada translator, we made our first attempt to translate and execute the target code in SDVS. We were immediately beset by a great number of heretofore undetected bugs in SDVS. SDVS had never been stressed with such a large Ada program before. We estimate that more than half of our time on this project was spent in fixing bugs or in attempting to deal with time and storage problems. As yet, the latter are not totally resolved.

After the bugs were corrected, we were able to complete a proof of correctness in SDVS of a program processing exactly three commands. The scheduler first made three calls to CMD_IN_HANDLER, then a call to BUILD, and finally a call to PROCESS_MSG. The input condition asserted that the input consisted of three commands encoding an "AP propagation time" data-structure message to be written in EEPROM. The output condition asserted that the message stored in EEPROM corresponded, in a manner stipulated by the MSX documentation, to the input commands. Even for such a minor proof, the SDVS proof trace was over 350 pages long.

We next completed the *inf_seq_of_2_msg_types* proof. Its input condition specified an input consisting of an infinite sequence of blocks of commands, each block encoding one of two types of data-structure messages: an "AP propagation time" message (message ID 10) or a "filter loss of track rules" message (message ID 22). In an infinite cycle, the scheduler called CMD_IN_HANDLER precisely the number of times needed to retrieve the next block from the input, then BUILD, and finally PROCESS_MSG. The output condition asserted that for each block of commands in the input, there is a message in the output corresponding to the input block in the manner stipulated by the MSX documentation. Furthermore, the correspondence of blocks of commands to messages is one-to-one, onto, and order-preserving.

It would seem that *inf_seq_of_2_msg_types* could be trivially generalized to a proof whose input condition specifies an input consisting of an infinite sequence of blocks of commands, each block encoding *any* one of the 61 data-structure messages and whose output condition is amended in the obvious way. (For future reference, we refer to this proof as the "*inf_seq_of_61_msg_types*" proof.) However, our attempts to complete *inf_seq_of_61_msg_types* were thwarted by the amount of time required for its completion: the system on which we ran it never stayed up long enough to complete it. We estimated it would take at least seven days for the proof to terminate.

The major problem is that in proving that the nth block of commands is processed correctly, each of the 61 possible cases for this block must be considered separately (at least in our approach). Each case adds to the execution time of the proof. One of our solutions was to run many separate but similar proofs. In each proof we considered only a few of the cases for the nth block of commands, proved that one case, and deferred the proof for the other cases. But we ran into a problem. Because four of the messages were unusually long, the correctness proof for each one of these messages required more storage than that allowed by the SDVS image (about 70 megabytes) and even more storage than that allowed by the SDVS image we created specifically for the MSX verification project (300 megabytes). We were thus unable to verify the correctness for these four messages.

In our attempt to execute the *inf_seq_of_61_msg_types* proof, we had to implement three new proof rules in SDVS: *repeat*, *loop*, and *if.* These were necessitated by the sheer size of the proof and are discussed in the next section. As we said in the previous page, we were never able to complete *inf_seq_of_61_msg_types*, because the system never stayed up long enough for us to complete it. But we were able to complete 57 of the cases in separate proofs.

Towards the latter part of the project, we realized that many of our time and storage problems would be solved if we proved adalemmas for some of the key subprograms in BUILD and in CMD_IN_HANDLER. We were able to complete adalemmas for two of the main procedures in BUILD, and these are described in Section 6.

3 Enhancements Made to SDVS for the MSX Verification Project

There are two main types of enhancements we made to SDVS for the MSX verification project: enhancements to the SDVS Ada environment and the addition of three proof commands not specific to that environment.

We first list and comment on the SDVS Ada environment enhancements. The first five in the list are standard Ada constructs (see [5]), whereas the others are improvements to the SDVS Ada proof environment.

(i) Integer subtypes:
One of the important aspects of the addition of integer subtypes to the Ada translator is that an assignment to an integer subtype object is allowed by SDVS only if the value assigned to the object is within the range of the subtype.

(ii) Integer definition types:
The restrictions on assignment are similar to those for integer subtypes: an assignment to an integer definition type object is allowed by SDVS only if the value assigned to the object is within the range stipulated by its type definition.

(iii) Type conversions:
These conversions are allowed only for integer numeric types (integer types and integer definition types).

(iv) UNCHECKED_CONVERSION:
We have added UNCHECKED_CONVERSION only for integer numeric types.

(v) Length representation clauses:
These have been implemented only for integer numeric types. The length stipulated for these types must be great enough to represent the type's stipulated range.

(vi) Markpoints:
These are used to mark Ada statements. We have added this marking capability to SDVS to allow the user to *go* (execute symbolically) to a specific marked point in the program. The user sets a mark in an Ada comment line just before the statement being marked, using the notation "--@" (no spaces):

```
--@ foo
x := 1;
```

During symbolic execution, this yields ".pc = at(foo)" (where "pc" is the program counter) at the point where the state delta(s) representing the marked statement become usable. Despite this fact, the user will not see "#pc = at(foo)" appear explicitly in the postcondition of the previously applied state delta. Any Ada statement can be marked. Note that the marked program remains acceptable to an Ada compiler; the marked points are treated as comments by Ada compilers. A mark can be turned into a regular (uninterpreted in SDVS) comment simply by inserting a space between the "--" and the "@", or by beginning the whole line with an extra pair of hyphens (even a single hyphen will do, so long as it's not followed by a space).

(vii) *adasubprogenv*:
This query command is quite useful in connection with the Ada offline characterization facility. It displays the mapping of fully qualified program names to uniquely qualified place names for all places constituting the environment for the proof of an *adalemma* about a subprogram.

(viii) Improved handling of Ada array constants:
The initialization of constant arrays in SDVS requires inordinately large storage and is extremely time-consuming. We had to add special Lisp code to handle these initializations. This addition is experimental and is not incorporated into SDVS 12. In our proof, we load the file "array_constant.lisp" to perform the initialization of constant arrays. These enhancements will be added to a future version of SDVS when the implementation is more robust.

(ix) Bugs:
Some of the things we have been calling "bugs" were in fact restrictions on the SDVS Ada translator of which we were not aware. For example, the translator did not allow the following declaration in the package SERIAL_DIG_CMDS:

```
procedure RET_CMD_BUF_AND_STATUS
             (Cmd_Buf               : out CMD_BUF_TYPE;
              Cmd_Status_Buf        : out CMD_STATUS_BUF_TYPE;
              Cmd_Count             : out INTEGER;
              FIFO_Not_Empty_Count  : out INTEGER;
              IO_Status_Word        : out GLOBAL_TYPES.WORD;
              Lost_Cmd              : out BOOLEAN);
```

The reason was that the package also contained the object declaration
Cmd_Buf : CMD_BUF_TYPE;
We removed limitations such as this as they arose in the course of the project.

In the course of our proofs, we realized that the addition of three new proof commands to SDVS would obviate a lot of repetitive work. These three commands are *loop*, *repeat*, and *if*. Since we have not tested them extensively, we have not added them to SDVS 12. These commands enabled us to write a "meta-level" proof; instead of writing 57 subproofs for the 57 cases in the MSX proof, we wrote one meta-level proof for all these cases.

The *loop* command has the syntax "UNTIL <condition> WITH <proof-name>." It is effectively a generalization of the *go* command, except that instead of simply executing *apply* commands until <condition> is true, it repeatedly interprets (executes) the proof <proof-name> until the condition is true.

The *repeat* command has the syntax "<proof-name> ITERATING ON <iteration-variable> FROM <lower_bound> TO <upper_bound>." It allows the user to write a parameterized proof (that is, a proof which involves the <iteration-variable>) and then to execute the proof <proof-name> successively with each value in the range from <lower_bound> to <upper_bound> substituted for <iteration-variable>. The formulas <lower_bound> and <upper_bound> must simplify to integer values.

The *if* command has the syntax "IF <condition> THEN <proof-name>." This command interprets the proof <proof-name> if <condition> is true, and does nothing otherwise.

4 Modifications to the Code

In this section we list the extensive modifications we made to the target software. The great majority of these modifications were necessary either because the SDVS Ada translator does not handle Ada tasking or because parts of the target software were written in 1750A assembly. The entire program appears in Appendix A. We have marked every statement that we deleted or replaced in the SERIAL_DIG_CMDS and APP_MSGS packages.

1. Ada *with* clause:
   Our original intent was to implement the *with* clause into the SDVS Ada translator. During our work on the design of the implementation, we realized the implementation would take a great deal of time that would be better spent on the other Ada constructs, since we could achieve part of the effect of the *with* clause by simply ordering the packages in the main program in the "right" order. In many cases, if the *with* clause were not used when needed in the original software, it would not be possible to compile the packages. We do not claim that the way we dealt with this clause is not prone to error. However, we don't think our solution allowed us to prove the correctness of code (under the circumstances stipulated by our input condition) that was in fact incorrect.

2. Ada *tasks*:
   We have probably belabored this point: SDVS does not currently handle tasking. We declared the tasks as procedures and deleted any Ada construct that is specific to tasking. Thus, we deleted the simple loops in the three procedures that were originally tasks as well as the pertaining exits. A simple loop requires an infinite execution of its body, unless an exit in the loop intervenes. Thus, the purpose of a simple loop in a task is to ensure that the task executes infinitely often. Once we changed a task to a procedure and deleted the simple loop in its body, the procedure's infinite execution could be ensured only by the scheduler we developed. We had to delete the simple loops in these procedures, because otherwise, once one of these procedures was called by the scheduler, it would never relinquish control of the execution to the others. The scheduler now performs the function of the simple loops, that is, to execute the procedures repeatedly.

   We also deleted the exception handlers: they were used only for tests involving the tasks.

   We deleted calls to the Tartan-supplied ARTCLIENT that is used in the code to enter and leave critical sections of the tasks and the interrupt-driven procedure.

3. Ada *pragma*:
   We deleted all pragmas. As we noted in [1], pragmas are instructions to the compiler and are generally not meant to change the semantics of an Ada program. In almost all instances, the pragmas in the target software involved either tasking or interfacing with the 1750A assembly code. (The pragmas we deleted are *elaborate*, *priority*, *pack*, *Foreign_Body*, and *Linkage_Name*.)

4. INTRINSICS and INTRINSIC_FUNCTIONS:
   We deleted all references to the Tartan-supplied INTRINSICS package. The INTRINSIC_FUNCTIONS package consists of generic instantiations of generic functions appearing in the INTRINSICS package. We characterized the instantiations we needed in the target code (AND_I, AND_WORD, AND_BYTE, and OR_I) by adalemmas.

5. Embedded calls to 1750A code:
   We enumerate and comment upon the instances of calls to 1750A assembly subprograms in target code.

   - In CMD_IN_HANDLER, a call is made to the ASM_UTILITY.CHECK_PARITY function to obtain the parity of a command. We provided a trivial Ada body for the parity function that always returns good parity (*true*). This solution is not entirely satisfactory because it does not allow the possibility of checking the code to determine what happens if a command does not have good parity. Our original intent was to give an offline characterization of CHECK_PARITY, allowing the possibility of returning *false* in certain cases. We did not have the time to do this.
   - CMD_IN_HANDLER services an interrupt indicating that there is a command to be retrieved from the FIFO buffer. It obtains the command by the following call:

     ```
     MEMORY_MANAGER.READ_FIFO(Cmd_In_FIFO_Addr,
                              Cmd_Unpacked'address,
                              Cmd_Size);
     ```

     (MEMORY_MANAGER is written in 1750A assembly.) We chose this to be the point at which the target code obtains input, and substituted the above call to MEMORY_MANAGER.READ_FIFO by a "for" loop that gets four bytes for the command from the standard input.
   - In PROCESS_MSG, a call is made to MEMORY_MANAGER.CONV_TYPE to assign the value of New_App_Msg(2) (an integer) to Temp_Word_1 (a word). The body of CONV_TYPE is written in 1750A. We replaced this call by a type conversion: `Temp_Word_1 := GLOBAL_TYPES.WORD(New_App_Msg(2));`

     In MANAGE_MSG_RETRIEVAL, another call is made to MEMORY_MANAGER.CONV_TYPE to convert each word stored by an array of words to an integer. We replaced this call by the following Ada type conversion:

     ```
     for index in 1 .. Num_Words_Curr_App_Msg loop
        App_Msg_Out(index) := INTEGER(App_Msg_Q(Q_Head)(index));
     end loop;
     ```
   - Again, in CONTINUE_BUILD, a call is made to MEMORY_MANAGER.CONV_TYPE to convert an integer to a word. We replaced this call by an Ada type conversion.

6. Hexadecimal Notation:
   We have replaced all hexadecimal literals by equivalent decimal literals.

7. Two-dimensional arrays:
   We have replaced all two-dimensional arrays with one-dimensional arrays of one-dimensional arrays.

8. Ada *others* initialization construct:
   Originally, we intended to add the *others* initialization construct to the SDVS Ada translator, but we did not have time to do it. However, in every case it is used in the target code, it is used to initialize arrays or records with values that are never actually used within this code. We simply deleted these initializations. In SDVS, these objects would simply have unknown symbolic values initially. Since nothing is known about these values, nothing could be proved from the assumption that they are unknown.

9. Ada *WRITELN* and *WRITESTRING*:
   We deleted all *WRITELN* and *WRITESTRING* constructs, which are not part of the standard Ada definition but are accepted by the Tartan compiler. These constructs appear in the target software for testing purposes only.

10. Ada *POSITIVE* type:
    This predefined subtype is used in the package CMDS_TYPES. We replaced these occurences by equivalent code. For example, we replaced the declaration

    ```
    subtype INDEX_SUBTYPE  is POSITIVE range 1..App_Msg_Max;
    ```

    by the declaration

    ```
    subtype INDEX_SUBTYPE  is INTEGER range 1..App_Msg_Max;
    ```

11. Unconstrained array types:
    There is one unconstrained array type, APP_MSG_OUT_TYPE, defined in CMDS_TYPES. We redefined this type to be a constrained array type with the maximum range possible:

    ```
      App_Msg_Max            : constant INTEGER := 80;
    --SDVS comment: The predefined type POSITIVE is not implemented in SDVS.
    --SDVS delete:  subtype INDEX_SUBTYPE  is POSITIVE range 1..App_Msg_Max;
      subtype INDEX_SUBTYPE  is INTEGER range 1..App_Msg_Max;--SDVS replace
    --SDVS comment: SDVS does not currently handle unconstrained array types.
    --SDVS delete (unconstrained array):  type APP_MSG_OUT_TYPE  is
    --SDVS cont:         array (INDEX_SUBTYPE range<>) of INTEGER;
      type APP_MSG_OUT_TYPE  is array (1 .. App_Msg_Max) of INTEGER; --SDVS replace
      Load_Msg_Max           : constant INTEGER := 127;
      type LOAD_MSG_TYPE     is array (1..Load_Msg_Max) of INTEGER;
    ```

12. Array slices
    In PROCESS_MSG, a slice of the array New_App_Msg is written to EEPROM by the call

```
-- Strip off New_App_Msg(1) which contains opcode,
-- the contents of data structure (including checksum)is
-- stored in New_App_Msg(2) thru (Data_Struc_Length+1).
--New_App_Msg(1..Data_Struc_Length) :=
--                 New_App_Msg(2..Data_Struc_Length+1);
EEPROM_DATA.STORE_DATA_STRUC_IN_EEPROM
          (New_App_Msg(2..Data_Struc_Length+1),
           Data_Struc_ID,
           Data_Struc_Length);
```

The slice is, in fact, the entire message. Since the SDVS Ada translator does not handle array slices, we have replaced the above call by

```
EEPROM_DATA.STORE_DATA_STRUC_IN_EEPROM
         (New_App_Msg,
          Data_Struc_ID,
          Data_Struc_Length);
```

The procedure STORE_DATA_STRUC_IN_EEPROM, whose body we wrote, simply outputs the array slice from 2 to `Data_Struc_Length + 1`.

5 Proofs for an Infinite Sequence of Data Structure Messages

We have finished a proof that MSX_PROGRAM_FINAL_VERSION, a suitably modified version of the original program, correctly processes infinite sequences of 57 of the 61 data-structure messages. This section contains a discussion of the proof and Appendix B contains the complete proof.

Before we discuss the proof itself, we must first note a few facts about input and output in SDVS for Ada programs in general, and input and output in the MSX_PROGRAM_FINAL_VERSION program in particular. We also describe the scheduler for this program, i.e., its body, in the following section.

5.1 Preliminaries to program input and output

For each program translated by the SDVS Ada translator, SDVS declares two objects, *stdin* and *stdout*, to be one-dimensional unbounded arrays of polymorphic type. *Stdin* is the standard input array: every value input via a call to the function *get* is obtained from the standard input array. The position in the array of the item obtained is determined by the order of the calls. For example, if *get(x)* is the first call to *get* in a program, then object x is assigned the value stored in *stdin[1]*. The standard input array is constant throughout the execution of the program. The standard output array, *stdout*, has a similar function, although unlike *stdin* it changes during the execution of a program when the function *put* is called. In SDVS, assertions about the input or the output of a program are assertions about the current values of the standard input array and the future values of the standard output array.

In the design of the target software, a byte is eight bits long; a word is sixteen bits long; and a command consists of two words. In the actual Ada code, bytes and words are represented by integers constrained to specific ranges. Commands are represented either as arrays of four bytes or as records with four fields, each field corresponding to one byte. Although bytes (words) have an integer parent type, they encode sequences of zeros and ones that are eight (sixteen) bits long. For example, if the integer value of the byte B is seven, then B encodes (i.e., contains) the bit sequence $< 00000111 >$.

In the target software, CMD_IN_HANDLER obtains a command by the call

```
MEMORY_MANAGER.READ_FIFO(Cmd_In_FIFO_Addr,
                         Cmd_Unpacked'address,
                         Cmd_Size);
```

Cmd_Unpacked is an array of four bytes, and MEMORY_MANAGER.READ_FIFO is a procedure with a body written in 1750A assembly code. We have substituted this call by the following:

```
for i in 1 .. 4 loop
  get(cmd_unpacked(i));
end loop;
```

Thus, every call to CMD_IN_HANDLER gets four elements (bytes) from the standard input array.

In the target software, PROCESS_MSG "writes" a message to EEPROM by the call

```
EEPROM_DATA.STORE_DATA_STRUC_IN_EEPROM
                          (New_App_Msg(2..Data_Struc_Length+1),
                           Data_Struc_ID,
                           Data_Struc_Length);
```

New_App_Msg, an array of words, is the processed message.

In our modified code, we have changed this call to

```
EEPROM_DATA.STORE_DATA_STRUC_IN_EEPROM
                          (New_App_Msg,
                           Data_Struc_ID,
                           Data_Struc_Length);
```

and decided to stop the message processing at this step.[5] At this point, the message has been built; we have changed the body of STORE_DATA_STRUC_IN_EEPROM so that it simply outputs the message as follows:

```
procedure STORE_DATA_STRUC_IN_EEPROM
       (Array_Of_Words           : in     CMDS_TYPES.APP_MSG_OUT_TYPE;
        Data_Struc_ID            : in     INTEGER;
        Data_Struc_Length        : in     INTEGER) is
begin
 for i in 2 .. Data_Struc_Length + 1 loop
   put(Array_Of_Words(i));
 end loop;
end STORE_DATA_STRUC_IN_EEPROM;
```

[5]We decided to verify the load to EEPROM up to this point only, since the message has been built at this point, and the machinery for the "write" to EEPROM has been initiated. The execution that follows in the original MSX software shortly leads to the invocation of procedures written in 1750A code.

Thus every call to STORE_DATA_STRUC_IN_EEPROM puts a message in the standard output array, word by word.

Having discussed the points at which calls to *get* and *put* occur in the modified software, we may now present the order of execution of the modified interrupt-driven procedure and the modified tasks. The execution order is determined by the body of MSX_PROGRAM_FINAL_VERSION:

```
begin
 --@ loop_begin

  loop
     SERIAL_DIG_CMDS.CMD_IN_HANDLER;
     for i in  2 .. APP_MSGS.RET_MSG_LENGTH(SERIAL_DIG_CMDS.RET_MSG_ID) loop
        SERIAL_DIG_CMDS.CMD_IN_HANDLER;
     end loop;
     APP_MSGS.BUILD;
     APP_MSGS.PROCESS_MSG;
  end loop;

end MSX_PROGRAM_FINAL_VERSION;
```

Each iteration of this loop accomplishes four things. In order, they are as follows:

- The first command of the next message is read (call to SERIAL_DIG_CMDS.CMD_IN_HANDLER).
- The remaining commands of the next message are read (*for* loop).
- The data for the newly read message are extracted and put into a new format consisting of words (call to APP_MSGS.BUILD).
- These words are output (call to APP_MSGS.PROCESS_MSG).

A property of this loop crucial to the proof is that it processes exactly one message block on each iteration.

5.2 The correctness assertion for MSX_PROGRAM_FINAL_VERSION

The correctness assertion for MSX_PROGRAM_FINAL_VERSION is the state delta *infinite_sequence_data_structure_messages.sd.* It is the formal equivalent of the informal correctness assertion "the Ada program MSX_PROGRAM_FINAL_VERSION correctly partitions and reformats an infinite input stream of bytes into its constituent messages".

```
(defsd infinite_sequence_data_structure_messages.sd
    "[sd pre: (formula(finale.sd),
               formula(input_places_disjoint.sd),
               formula(output_places_disjoint.sd),
               ada(msx_program_final_version.a),
               formula(checksum_first_k_bytes_message_n_definition),
               formula(checksum_definition),
               formula(msg_in_lh_definition),
               formula(msg_out_lh_definition),
               formula(msg_input_begins_at_definition),
               formula(msg_output_begins_at_definition),
               formula(allowed_message_ids),
               formula(input_condition))
      comod: (all)
        mod: (all)
       post: (formula(output_condition)) ]")
```

The first three formulas in the precondition of this state delta (*finale.sd*, *input_places_disjoint.sd*, and *output_places_disjoint.sd*) are state deltas that are used in the proof of *infinite_sequence_data_structure_messages.sd.* Each of these three state deltas has been proved in SDVS, and their proofs are included in Appendix B. Usually such state deltas are written as lemmas and do not appear in the precondition, but these formulas contain quantifiers and the current SDVS implementation cannot handle lemmas that contain quantifiers. In normal circumstances, we would have simply proven these facts in the same SDVS session with the main proof, but the way we conducted the final proof (explained in more detail later) meant that we would have had to run the proofs of these state deltas (which take a significant amount of time) many times. Since we were already concerned about the amount of time the main proof was going to take, we decided to prove these once and include them in the precondition of the correctness assertion rather than reprove them many times.

The next part of the precondition (*ada(msx_program_final_version.a)*) is a predicate that corresponds to the program MSX_PROGRAM_FINAL_VERSION. The formula *input_condition* in the precondition specifies the input condition for this program: it depends on the formulas *checksum_first_k_bytes_message_n_definition*, *checksum_definition*, *msg_in_lh_definition*, *msg_out_lh_definition*, *msg_input_begins_at_definition*, and *allowed_message_ids.* The formula *output_condition* in the postcondition specifies the output condition: it depends on the formulas *msg_out_lh_definition* and *msg_input_begins_at_definition.*

The formula *allowed_message_ids* and those formulas whose names end in *_definition* are used in the correctness assertion only to help define the input condition and output condition, and are strictly speaking unnecessary: that is, we could have written the correctness assertion without them. We feel, however, that they aid greatly in simplifying the proof and in making it understandable. In the next two subsections, we will discuss first the formulas used in the input condition, and then the formulas used in the output condition.

5.2.1 The input condition

In this subsection we will discuss in detail the input condition for the program. As mentioned earlier, the main formula used to specify the input condition is the formula *input_condition* below (this formula is taken directly from page 146 in Appendix B).

```
(defformula input_condition
    "forall x (forall z (x ge 1 -->
                ; The first byte is always 1
                  ((z = 1 --> .stdin[msg_input_begins_at(x) + (z-1)] = 1) &

                ; The second byte is always 2
                  (z = 2 --> .stdin[msg_input_begins_at(x) + (z-1)] = 2) &

                ; The third byte is the message identifier
                  (z = 3 --> .stdin[msg_input_begins_at(x) + (z-1)] = msg_id(x)) &

                ; As for the remaining bytes...
                  ((4 le z & z le  msg_in_lh(x)) -->

                    ; If it's the first byte of a command, it's an eight
                      ((z mod 4 = 1 -->
                         .stdin[msg_input_begins_at(x) + (z-1)] = 8) &

                    ; Otherwise, we only know it's a byte
                      (z mod 4 ~= 1 -->
                         is.byte(.stdin[msg_input_begins_at(x) + (z-1)])))) &

                ; First checksum byte is the high bits of the checksum word
                ; for message x
                  ((z =  ((8 * msg_out_lh(x) - 2) / 3)) -->
                         .stdin[msg_input_begins_at(x) + (z-1)] =
                             high.bits(checksum(x))) &

                ; Second checksum byte is the low bits of the checksum word
                ; for message x
                  ((z =  ((8 * msg_out_lh(x) + 2) / 3)) -->
                         .stdin[msg_input_begins_at(x) + (z-1)] =
                              low.bits(checksum(x))))))")
```

This formula (in conjunction with other formulas) specifies an infinite sequence of data structure messages: the embedded comments show how it corresponds to the informal specification of a data-structure message given in [1]. Examination of this formula reveals that it depends on the macros *is.byte(x)*, *high.bits(x)*, and *low.bits(x)*, and on the functions *msg_input_begins_at*, *msg_in_lh*,*msg_id*, and *checksum*. We will first explain the macros, then the functions *msg_input_begins_at*, *msg_in_lh*, and *msg_id*, and finally the function *checksum*. In the course of explaining these functions (and other functions used to define them), we will explain all of the functions in the precondition.

First the macros. The macro *is.byte(x)* is true when x is between 0 and 255, inclusive, false otherwise. The macros *high.bits(x)* and *low.bits(x)* are used to represent the most and least significant eight bits of a word, respectively.

We will now discuss those functions used in the input condition. To clarify the following discussion, we have provided Table 1, which illustrates the values of the functions for a message sequence whose first three messages in order have identifiers 22, 10, and 1. All of the constituent bytes of the first two messages, and the first six of the third, are shown in Table 1. This table also includes a condensed display of the nth message to illustrate the values of these functions for input value n.

The function *msg_input_begins_at* plays an important role in specifying the input. As the name suggests, the value of this function at x is the location of the byte where the xth message begins. Thus the first byte of the xth message is *.stdin*[*msg_input_begins_at*(x)], the second is *.stdin*[*msg_input_begins_at*(x) + 1], etc. This function is defined in the following way:

```
(defformula msg_input_begins_at_definition
    "forall x (msg_input_begins_at(1) = 1 &
              ((x gt 1) -->
                (msg_input_begins_at(x) =
                  msg_input_begins_at(x-1) + msg_in_lh(x-1))))")
```

This definition is based on the observation that the first byte of the xth message is one plus the sum of the lengths of the preceding messages. We can see this by expanding the definition above at x:

$$\mathit{msg\_input\_begins\_at}(x) =$$

$$\mathit{msg\_input\_begins\_at}(x-1) + \mathit{msg\_in\_lh}(x-1) =$$

$$(\mathit{msg\_input\_begins\_at}(x-2) + \mathit{msg\_in\_lh}(x-2)) + \mathit{msg\_in\_lh}(x-1) =$$

$$\vdots$$

$$(\ldots((1 + \mathit{msg\_in\_lh}(1)) + \mathit{msg\_in\_lh}(2)) + \ldots + \mathit{msg\_in\_lh}(x-2)) + \mathit{msg\_in\_lh}(x-1)$$

The function *msg_in_lh* used above is defined in the formula *msg_in_lh_definition* on page 142 in Appendix B. This definition is based on the specification of the message input lengths as determined by the message identifier. The formula *msg_in_lh_definition* employs the function *msg_id*(x), which as we can see from the formula *input_condition* above, returns the same value at x as the message identifier for message x (i.e., the third byte of message x). An important part of the input condition is the formula *allowed_message_ids* (given on

Table 1: Example Input Sequence

	Byte	Value
Block 1	1 (= $msg\_input\_begins\_at(1)$)	1
	2	2
	3	22 (= $msg\_id(1)$)
	4	$d_{1,1}$
	5	8
	6	$d_{1,2}$
	7	$high.bits(checksum(1))$
	8	$low.bits(checksum(1))$
Block 2	9 (= $msg\_input\_begins\_at(2)$)	1
	10	2
	11	10 (= $msg\_id(2)$)
	12	$d_{2,1}$
	13	8
	14	$d_{2,2}$
	15	$d_{2,3}$
	16	$d_{2,4}$
	17	8
	18	$high.bits(checksum(2))$
	19	$low.bits(checksum(2))$
	20	spare
Block 3	21 (= $msg\_input\_begins\_at(3)$)	1
	21	2
	22	1 (= $msg\_id(3)$)
	23	$d_{3,1}$
	24	8
	25	$d_{3,2}$
	$\vdots$	$\vdots$
Block n	$msg\_input\_begins\_at(n)$	1
	$msg\_input\_begins\_at(n) + 1$	2
	$msg\_input\_begins\_at(n) + 2$	$msg\_id(n)$
	$msg\_input\_begins\_at(n) + 3$	$d_{n,1}$
	$msg\_input\_begins\_at(n) + 4$	8
	$msg\_input\_begins\_at(n) + 5$	$d_{n,2}$
	$\vdots$	$\vdots$
	$msg\_input\_begins\_at(n) + msg\_in\_lh(n) - 1$	$low.bits(checksum(n))$ or spare
Block $n+1$	$msg\_input\_begins\_at(n+1)$	1
	$\vdots$	$\vdots$

page 141 of Appendix B), which constrains the value of *msg_id*(x) (and hence the third byte of message x) to be a legitimate message identifier.

Finally, the function *checksum* (defined on page 146 in the Appendix) represents the result of the program's calculation of a message's checksum. This function is defined with the help of two other functions: *sdvs_xor_word*(x, y), which corresponds to the Ada XOR_WORD(X,Y) function; and *checksum_first_k_bytes_message_n*(n, k), which uses the *sdvs_xor_word* function and represents the result of the checksum calculation on the first k words of the nth message.

The position of the checksum bytes is given in the formula *input_condition* in terms of the function *msg_out_lh*. This function and the function *msg_output_begins_at* are defined for output in the same way that are *msg_in_lh* and *msg_input_begins_at* are defined for input. We will discuss *msg_out_lh* and *msg_output_begins_at* in the next section.

5.2.2 The output condition

The output condition for the correctness assertion is given by the formula *output_condition* below:

```
(defformula output_condition
    "x ge 1 --> (forall  z (( 1 le z & z le msg_out_lh(x))
                  --> #stdout[msg_output_begins_at(x) +(z-1)]
                         = mk.word(.stdin[msg_input_begins_at(x)
                                 + ((8 * z - 5) / 3)],
                               .stdin[msg_input_begins_at(x)
                                 + ((8 * z - 1) / 3)])))")
```

This formally expresses the relationship between the input and output given informally in [1], which in brief may be stated "the output for a message is obtained by extracting the actual data bytes from the input for that message and forming them into words". This formula depends on the macro *mk.word* and the functions *msg_output_begins_at* and *msg_out_lh*. The macro *mk.word(x,y)* is a function that gives the integer value of a word whose high bits are the byte x and whose low bits are the byte y. This macro is used to capture succinctly the operation of "...forming them into words", given in the quotation earlier in this paragraph.

As indicated earlier, the functions *msg_output_begins_at* and *msg_out_lh* (defined on pages 145 and 144, respectively) are the analogues of *msg_input_begins_at* and *msg_in_lh* for output. As with the input condition, for clarification we include a table (Table 2) of the output for the input given in Table 1. Comparing Table 1 and Table 2 shows that the input and output for the example sequence do in fact have the relationship stated in the quotation in the preceding paragraph, with the last word of the output for message n being the checksum for that message [i.e., *checksum*(n)].

Table 2: Example Output Sequence

	Word	Value
Block 1	1 (= $msg\_output\_begins\_at(1)$)	$mk.word(2, 22)$
	2	$mk.word(d_{1,1}, d_{1,2})$
	3	$checksum(1)$
Block 2	4 (= $msg\_output\_begins\_at(2)$)	$mk.word(2, 10)$
	5	$mk.word(d_{2,1}, d_{2,2})$
	6	$mk.word(d_{2,3}, d_{2,4})$
	7	$checksum(2)$
Block 3	8 (= $msg\_output\_begins\_at(3)$)	$mk.word(2, 1)$
	9	$mk.word(d_{3,1}, d_{3,2})$
	$\vdots$	$\vdots$
Block n	$msg\_output\_begins\_at(n)$	$mk.word(2, msg\_id(n))$
	$msg\_output\_begins\_at(n) + 1$	$mk.word(d_{n,1}, d_{n,2})$
	$\vdots$	$\vdots$
	$msg\_input\_begins\_at(n) + msg\_out\_lh(n) - 1$	$checksum(n)$
Block $n+1$	$msg\_input\_begins\_at(n+1)$	$mk.word(2, msg\_id(n+1))$
	$\vdots$	$\vdots$

5.3 The proof of infinite_sequence_data_structure_messages.sd

In this subsection we will talk about the proof of the state delta *infinite_sequence_data_structure_messages.sd.* We will present here the proof given in Appendix B (which is there presented in a "bottom-up" fashion) in a "top-down" style, proceeding from a high-level view of the proof through increasing levels of detail.

As we have mentioned before, MSX_PROGRAM_FINAL_VERSION has an infinite loop as its main body. Our proof strategy in the proof of *infinite_sequence_data_structure_messages.sd* (*main.proof* on page 197 of Appendix B) is to execute symbolically to the beginning of this loop and to then use loop induction. As is typical with loop induction, we chose a loop invariant weak enough to be proved after symbolically executing through the loop, but strong enough to imply the goal when the induction was finished. In our case, the invariant of the loop induction is the following:

$\vdots$

```
comment   "Starting induction...",

induct on:    n
  from:       1
```

```
to:          x + 1
invariants:
  (
   ;Needed for covering information

   pcovering(.msx_program_final_version,old_universe),

   ;Variable values

   .build_in_progress = false,
   .cmd_count = 0,
   .lost_cmd = false,
   .app_msg_counter = 0,

   ;Queue variables and bounds

   .q_head
      = 1 + .q_tail mod .app_msg_q_size,
   .q_head ge origin(app_msg_q),
   .q_head
      le (origin(app_msg_q) +
          range(app_msg_q)) - 1,
   .q_tail mod .app_msg_q_size + 1
      ge origin(app_msg_q),
   .q_tail mod .app_msg_q_size + 1
      le (origin(app_msg_q) +
          range(app_msg_q)) - 1,

   ;State delta at beginning of loop

   formula(loopsd),

   ;Input/output counters are where they should be

   msg_input_begins_at(n) = .stdin\ctr,
   msg_output_begins_at(n) = .stdout\ctr,

   ;Output condition is true after execution of loop when
   ;n = x. This is stated in this way due to efficiency
   ;considerations (i.e., the more straightforward way
   ;introduces many new places - see report for more details).

   n = x + 1 --> formula(output_condition))
```

$\vdots$

The crucial formula in the invariant above is the last one. When the loop induction is completed, the upper bound of the loop $x+1$ is substituted for n in this formula, and thus $x+1 = x+1 \rightarrow formula(output\_condition)$ is true after the induction is finished.It is then easy to proceed to the proof of $formula(output\_condition)$. Instead of this formula, we originally tried $formula(output\_condition)$ with n substituted for x, but we found that using this in the invariant caused the prover to slow down significantly and we aborted the proof before it finished.

Whenever SDVS encounters a new place, it must add information to its database about whether changes to that place cause changes to any of the places already in the database. Both the operation of adding a new place to the database and the operation of checking to see what other places are affected when a given place is changed can be time-consuming when there are many places and some of the places are not known to be disjoint, i.e., changes to these variables may cause changes to other variables. Using $formula(output\_condition)$ with n substituted for x in the invariant of the **induct** command introduced elements of *stdin* and *stdout* that are not necessarily disjoint from elements of these arrays introduced earlier in the proof (specifically, via the postcondition of the correctness assertion, which is simply $formula(output\_condition)$) and thus was responsible for the proof proceeding slowly. The formula we actually used avoids this problem, as it contains exactly the same elements of *stdin* and *stdout* as $formula(output\_condition)$.

As with any loop induction, there is a base case and a step case: these are handled in the main proof above by *base_case.proof* and *step_case.proof*, respectively. The base case proof is short and straightforward, as most of the goals are automatically proved by SDVS. The step case is proved by doing a case split on all possible values of $msg\_id(n)$, and then using the same meta-level proof *every_case.proof* in each case . Aided by the new proof commands *loop*, *repeat*, *if* (described in an earlier section) and *letq* (which allows quantified formulas to be given more meaningful names), we were able to write such a meta-level proof because of the similarity of the proof for each case.

After system crashes thwarted our attempts to complete the week-long proof, we decided that the full proof would be done in a number of separate SDVS runs rather than a single run. This was by done using different versions of *step_case.proof* (these begin on page 168 of Appendix B) when running the main proof. Each of these versions carries the proof through only for certain message identifiers and defers the cases for other message identifiers. For example, the first version of *step_case.proof* proves correctness for the case $msg\_id(n) = 1$ and defers all the other cases. These versions cover all possible cases except for message identifiers 53, 54, 55, and 56. The messages with these identifiers are all substantially longer than any of the other messages, and because of the structure of the meta-level proof, the time and space requirements for each case vary directly with the length of the message treated in that case. SDVS ran out of storage when we tried to run proofs of these cases, in spite of efforts such as increasing the amount of space in the SDVS image and modifying the way in which garbage was collected.

In the proof *every_case.proof*, each case is further split into two cases, one in which n is less than x and one in which n is equal to x. Although this split entails symbolically executing through the loop twice, this proof was much faster than any that did not involve the case split. The reason for the slowness of the proofs without a case split is essentially the same as the reason (mentioned a few paragraphs ago) for the slowness of the proof with the original invariant: namely, in those proofs without the case split, the prover was not able to recognize the disjointness of various elements of *stdin* and *stdout*.

Both of these cases in *every_case.proof* have a common part:

```
                              ⋮

interpret produce_input_information,
interpret go_to_return_from_call_to_ret_cmd_buf_and_status,
interpret fix_array_values_after_call_to_ret_cmd_buf_and_status,
interpret handle_intrinsics_until_end_of_program,
interpret finish_off_non_output_goals,

                              ⋮
```

The above proof commands accomplish the following:

- Instantiate the formula *input_condition* for the appropriate range of values to provide information about the current message's input needed to execute symbolically through the loop, e.g. instantiate *input_condition* with 3 to inform SDVS that the third byte equals the message identifier (*produce_input_information*);
- Symbolically execute until the end of the procedure RET_CMD_BUF_AND_STATUS and then execute **notice** and **consider** commands to "remind" the prover of values that occurred in an array assignment (*go_to_return_from_call_to_ret_cmd_buf_and_status* and *fix_array_values_after_call_to_ret_cmd_buf_and_status*);
- Symbolically execute until the beginning of the loop is reached (*handle_intrinsics_until_end_of_program*);
- Prove all of those goals generated by the *induct* command that do not have to do with the output condition (*finish_off_non_output_goals*)

In the case where n is less than x, we also interpret the proofs *separate_input_places* and *separate_output_places* before the common part of the proof given above. These proofs provide SDVS with enough information to deduce that the problematic places that necessitated the case split are disjoint. In the other case (where n is less than x), we interpret the proof *finish_off_output_goal* to prove the goal generated by the induct command that involves the output condition.

The central part of *every_case.proof* is the subproof *handle_intrinsics_until_end_of_program*. Stripped of comments and timing commands, this proof is simply the proof command

```
loop until formula(loopsd) with loop_body
```

Thus this subproof interprets the subproof *loop_body* until *formula*(*loopsd*) is true, i.e., execution reaches the beginning of the loop. The proof *loop_body* symbolically executes until (1) the call of an intrinsic function, (2) after the call to MANAGE_MSG_RETRIEVAL, (3) at the program label *after_checksum_calculation*, or (4) at the beginning of the loop. When (1) occurs, we apply an adalemma for that Ada function. At (2), we again need to "remind" SDVS of lost array values. For (3), we have to prove that the checksum calculation yields the same value as the word formed from the checksum bytes. When (4) is true, the symbolic execution of the loop is finished, the interpretation of *handle_intrinsics_until_end_of_program* is finished, and we proceed to proving the goals.

This concludes the discussion of the proof of the correctness assertion. More detail about the proof is contained in Appendix B. It should be noted that in some sense, this proof proceeds by "brute force." By this we mean that rather than symbolically execute through the loop once and have a proof that takes into account the multiple computation paths through the loop for the different messages, we simply split it up into cases such that there is exactly one computation path (and hence a simpler, in fact a meta-level proof) for each case. Thus the way we have chosen to prove the correctness assertion involves many relatively simple proofs, each of which involves symbolically executing through the loop (and is thus time-consuming), rather than a single, more complicated proof that symbolically executes through the loop once.

One way to improve upon the approach in this section is to use adalemmas for the various procedures in the program. This approach is explored in the next section.

6 Adalemmas for the BUILD task.

6.1 Why adalemmas?

Adalemmas are in essence a special type of state delta that can replace a subprogram call during symbolic execution of the main program. Normally symbolic execution proceeds through the point of a call to a subprogram into the code of the subprogram itself. If, however, an adalemma for the subprogram has been proved, it can be invoked, and, in one step, symbolic execution moves ahead to the return from the call. The adalemma should express all the significant changes to the program state that might result from execution of the subprogram, so that it is irrelevant to the larger proof in progress whether the subprogram code was executed or the adalemma was invoked. Further, the postcondition of the adalemma would normally provide descriptive information about what the subprogram has accomplished that would otherwise have to be proved during symbolic execution of the subprogram code. For example, an adalemma for a sorting procedure might include in its postcondition a quantified statement asserting that the output array is sorted. Finally, if a subprogram is called at several places in a program, invocation of the adalemma replaces several possibly lengthy symbolic executions of the subprogram code. Thus, adalemmas offer significant efficiencies in symbolic execution.

However, efficiency is neither the only nor the most important consideration. The main purposes of adalemmas are as follows:

(1) To provide a vehicle for expressing and proving abstract properties of a subprogram.

(2) To permit modular construction of proofs for programs.

(3) To enable specifications to be stated within the scope of program declarations.

We shall briefly discuss each of these points.

(1) Program specifications are often stated for sections of programs, rather than for an entire main program. This is true for reasons of manageability, conciseness, intelligibility, the fact that program specifications are usually better stated for subprograms, and sundry other considerations. In the case of the portions of the MSX code under consideration in this project, a high-level specification might involve the behavior of interacting asynchronous tasks; it is unclear whether or how formal verification can address the timing considerations involved, so we are led to specifications for individual tasks. The typical situation is that one has a specification either for a subprogram or for some programming unit that can be recast as a subprogram for the purposes of formal verification. Adalemmas in SDVS are expressly designed to meet the needs of this situation.

Further, specifications for programs are usually set up with a considerable degree of abstraction. A program with a completely concrete specification and set of inputs can be verified merely by testing; there is no need for formal methods. But in the usual situation of infinitely many possible inputs, the specifications are necessarily either abstract or incomplete. Adalemmas facilitate the abstract statement of specifications for subprograms.

(2) Just as good mathematical proofs are usually developed through a series of lemmas and subsidiary arguments, and good programming style calls for structured and modularized code, so program verification requires modularization. Without this, proofs of any but the simplest programs become mired in a mass of detail, overwhelming the user and defying comprehension. Adalemmas make it possible to avoid such situations, and to build up proofs for complex programs in a step-wise fashion.

(3) Adalemmas enable specifications to be stated within the scope of program declarations. Without adalemmas, the claim of correctness of a program with respect to its specifications (which as always with SDVS is formulated as a state delta to be proved) must be formulated outside the scope of the program declarations. This is not without its merits, because then the program is treated as a "black box," and the verification covers the declarations in the code as well as the program structure. However, this makes it difficult to state things abstractly, since all parameters involved in the specification, possibly including large collections of constants, have to be asserted in a setting-up phase of the proof preceding the symbolic execution of the program. On the other hand, an adalemma can be formulated in a context in which reference to the program declarations is possible, because all declarations and initializations are automatically generated by the translator; this facilitates abstract treatment and usually increases the generality of the resulting proof.

This point is perhaps easier to see if we examine some of the earlier verification efforts with SDVS, prior to the availability of adalemmas. One of the first examples was a sorting procedure. Since the specification could not be stated for the procedure alone, it had to be stated for an enclosing main program that would actually test the procedure. This program would read in an array, call the procedure, and write out the result. The proof had to contain an induction for the input loop, and a second induction for the output loop, as well as quantified statements expressing the connection between the internal SDVS data structures and the *stdin* and *stdout* arrays. All of this really had nothing to do with verification of the target procedure, and obfuscated the proof. None of this was necessary once the adalemma facility was introduced.

As mentioned above, there are some conceptual advantages to treating the whole program from the outside—the proof isn't dependent upon the declarations—but it is arguable that declarations are neutral from the point of view of verification. There might be a mistake in program declarations that a "from the outside" proof would detect. But there could just as easily be a mistake in the part of that proof containing the assertions that correspond to the internal declarations. In the end, the (human) verifier must compare the printed specifications with the encodings, whether in the proof or the program text. Unlike declarations, the links between the detailed program structure and the specifications of functionality are relatively obscure, so formal verification has something to offer.

6.2 A proof strategy for BUILD

6.2.1 The BUILD process

In this section we give a reprise of the BUILD task. We assume general familiarity with the code, and our purpose here is merely to give an overview of BUILD to facilitate our description of a verification strategy. Tables 1 and 5 in [1] should illuminate parts of the discussion.

The BUILD task consists mainly of a nonterminating loop. At the beginning of the loop, some (0 or more) *commands* which have been received are placed in APP_MSGS.CMD_BUF by a call to SERIAL_DIG_CMDS.RET_CMD_BUF_AND_STATUS. The loop simply repeats if there are no new commands. Eventually, there will be some new commands, so the effect is essentially the same as if we regard each call to SERIAL_DIG_CMDS.RET_CMD_BUF_AND_STATUS as retrieving at least one command. BUILD processes one command at a time, i.e., per iteration, and for practical purposes we can view the incoming commands as an infinite stream. Each command has four bytes, the first byte indicating the type of command, and the remaining three containing the data. The commands supply raw data; the stream of commands must be parsed and the data repackaged into *application messages* meaningful to other modules of the tracking processor. This parsing and repackaging process is the function of the BUILD task. BUILD assembles incoming data in an internal buffer (the App_Msg array), and when a complete message has been constructed, it copies this into a FIFO queue of messages, the App_Msg_Q. Messages are removed from the App_Msg_Q by the task MANAGE_MSG_RETRIEVAL and passed to the task PROCESS_MESSAGE.

In this section we focus on the message-building process, independently of the interrupt-driven command reception process or the subsequent message retrieval and processing. That is, we are concerned solely with the BUILD task. The central aspect of this is to show that if the input stream of commands provides a correctly constituted message, then that message is eventually assembled by BUILD and placed in the App_Msg_Q. There are numerous other points that would merit verification, e.g. the handling of erroneous input, but in this report we restrict ourselves to verifying the basic functionality (assuming error-free input).

A command has four bytes. The first byte is a code indicating the type of command. Many op codes signify the type of message being initiated, and a few are used to identify the command as a continuation command for a previously initiated message. In the case of Data_Struc_Load messages, the op code of the initiating command is 1, while code 8 (Data_Load_Cmd) is used for the succeeding commands in this message type. Some types of messages are completed by only one command, i.e, they have at most two actual data bytes besides the op_code, and others require several commands to convey enough data for a complete message. We are here primarily interested in the latter type, and in particular, the Data_Struc_Load messages. These have op_code 1 and continuation code 8, and come in many sizes. The second byte of a Data_Struc_Load message contains information relating to the ultimate disposition of the message, but is not used in the build process. The third byte of a Data_Struc_Load message contains a *message_id*; associated with each message_id is a specific length for the message.

Let us consider the bytes of a message as being numbered sequentially. The first command for a message contains bytes 1 through 4, and each succeeding command supplies three more message bytes (the high-order byte of these four-byte commands contains the continuation code). The message bytes from the commands are copied and repackaged into an array of words, which is considered the message itself.

The code has two tables of constants:

> `TRKING_DATA_STRUC.Words_For_Data_Struc`, providing the actual number of words needed for a message, and
>
> `Cmds_For_Data_Struc`, providing the number of commands required to convey the message.

The message_id is the look-up index for both tables. Note that each command provides three bytes for a message, each word of a message holds two bytes, and the first message byte (the op_code) goes in the second byte of the first word. An n-byte message requires $(n-1)/3$ commands and $(n+1)/2$ words.

Most of the work of the `BUILD` task is accomplished by two procedures, `INITIATE_BUILD` and `CONTINUE_BUILD`. `INITIATE_BUILD` is called when a message-initiating command is received. It determines what type of message is to be built. In some cases, the entire message is available from the first command, so `INITIATE_BUILD` sends it directly to the `App_Msg_Q`. Other types of messages (`Data_Struc_Load`, `RAM_Mem_Load`, `EEPROM_Mem_Load`, `Mem_Dump_X_Frames`, and `Chg_Monitor_Loc`) require more than the four bytes available in the first command, so `INITIATE_BUILD` sets the `Build_In_Progress` flag to true, and puts the bytes of the first command into the `App_Msg` intermediate buffer. Each of the message types involves its own unique processing steps at some point, but we have decided to focus on the `Data_Struc_Load` messages (data-structure application messages) because they are critical to the mission and their length generally requires "building," so they provide a good example of the functioning of most parts of the `BUILD` task.

`CONTINUE_BUILD` is called when `Build_In_Progress` is true, and appends three more bytes to the message being built in the `App_Msg` buffer. In addition, if this is the last command with data for the current message, `CONTINUE_BUILD` copies the `App_Msg` to the `App_Msg_Q`, and then sets `BUILD` task global variables as required to reset the build process in preparation for the next message.

The following is a summary of the control flow for `BUILD`:

```
Start at loop beginning with some commands in  Cmd_Buf
Call  Initiate_Build to process the first command:
    Assume a  Data_Struc_Load message.
    Cont_Cmd <--- Data_Load_Cmd
    Determine  Num_Cmds_For_App_Msg, and  Num_Words_For_App_Msg
    via table look-up.
```

```
        Copy bytes 1 through 4 of the command into  App_Msg(2..4).
        Cmds_Rcvd  <---  1
        Build_In_Progress  <---  True
    While  Cmds_Rcvd < Num_Cmds_For_App_Msg - 1:
        Call  CONTINUE_BUILD to add the last three bytes of a
        continuation command to the message being built in the
        App_Msg buffer.
    Call  CONTINUE_BUILD for the last command for this message, to
        Add the last three bytes to  App_Msg,
        copy the completed application message from  App_Msg to
        the  App_Msg_Q,
        reinitialize task globals.
```

Many details have been omitted from this summary, but this is the general idea of the control flow. Our summary is intended to convey only the actual order of execution, not the precise syntactic structure of the program. In particular, there is no while-loop as shown in the summary; instead, almost the entire task consists of a single loop that extracts a command from the Cmd_Buf, then calls either INITIATE_BUILD or CONTINUE_BUILD depending on the value of the Build_In_Progress flag. CONTINUE_BUILD may be called even for a command that is not a continuation command; for example, a No_op_Cmd may be interspersed at any point during the stream of commands for some application message, and CONTINUE_BUILD will write it directly into the App_Msg_Q, but not abort the build in progress.

6.2.2 Proof strategy

As noted, SDVS does not currently support tasks. Indeed, problems of interaction between asynchronous concurrent tasks are generally intractible, and one should probably not expect any formal program-verification system within the current state of the art to deal with all aspects of such problems. However, it is certainly possible to formulate and prove meaningful statements about tasks. We shall outline a verification approach to the BUILD task.

BUILD is supposed to copy bytes from the Cmd_Buf, which are packaged as three bytes per command, and reassemble them into messages, with the completed message placed in the App_Msg_Q. This process is complicated by the facts that the Cmd_Buf need not contain at any one time a sufficient number of commands to complete a particular message, and that No_op commands may be interspersed with commands continuing some other message. We will deal with the latter complication by assuming that there are no No_op commands, and with the former by assuming that as a result of the call by BUILD to SERIAL_DIG_CMDS.RET_CMD_BUF_AND_STATUS, the Cmd_Buf contains enough messages to complete a message. The more general situations are probably not intractable for SDVS, but they are more than is appropriate for this first attempt at verification of BUILD.

Consider the beginning of the “for” loop in the BUILD task. A state delta expressing the essence of what this part of BUILD does should express the following:

Assume that control is at the beginning of the "for" loop, that the first command in the `Cmd_Buf`, namely `Cmd_Buf(1)`, is the first command of a `Data_Struc_Load` message, and that `Cmd_Count` is large enough that the complete message is contained within the commands in the buffer, i.e.,

```
.cmd_count ge .cmds_for_data_struc[.record(cmd_buf[1],third_byte)]
```

Then at a future time, control will again be at this point and the message will have been built and copied into the `App_Msg_Q`.

The state delta would include other information, such as the new values of variables, and the "future time" would be precisely specified.[6]

Some changes in the program syntax are necessary to accomodate SDVS and what we want to do. In the first place, SDVS does not handle tasks, so the `BUILD` task must be converted to a procedure. Secondly, we are interested in proving something about an internal fragment of `BUILD`, the "for" loop, so we must find a way to facilitate this within the requirements of SDVS. Enhancements of SDVS to permit proving lemmas about fragments of code are under consideration, but not yet in place. However, SDVS does have the ability to prove adalemmas about procedures. Any fragment of code which is a sequence of executable statements can be converted to a procedure call by the following artifice: Given a fragment of code

```
        --Start of code fragment
                .
                .
                .
                .
                .
                .
                .
        --End of code fragment
```

replace it by

```
DECLARE
        PROCEDURE Fragment IS
                BEGIN

                        --Start of code fragment
                                .
                                .
                                .
```

[6]A more elaborate treatment could drop the assumption that all the necessary commands were in the `Cmd_Buf`, and would consider conditions at the beginning of the outer main loop, instead of the inner "for" loop. In the actual implementation, commands are obtained asynchronously using hardware interrupts. Our test programs replace this by a simple call to "get," so the pending commands can be referred to as in the `stdin` array. Then the action of the routine `SERIAL_DIG_CMDS.RET_CMD_BUF_AND_STATUS` would be described as simply returning one or more commands from the input stream. Although all this seems feasible and not inordinately complicated, we believe the simpler form chosen above to be satisfactory for a first step.

```
                                        .
                                        .
                                        .
                                        .
                              --End of code fragment

                    END Fragment;
BEGIN

          Fragment;

END;
```

The result is essentially equivalent to the original code.[7] Whatever one may have wanted to prove about the original code fragment may now be formulated as an adalemma about the procedure `Fragment`. This adalemma can then be used in proofs about larger, enclosing procedures.

In the present case, we propose to prove a suitable adalemma for the "for" loop that is part of the `BUILD` procedure. Such an adalemma could then be used as an aid in proving other adalemmas about larger parts of `BUILD`.[8]

The main elements of the "for" loop are calls to `INITIATE_BUILD` and `CONTINUE_BUILD`. Our first steps are therefore to develop and prove adalemmas for these two procedures.

At the time of this writing, the proofs for the two procedures are complete, but the adalemma and proof for the for-loop are still under development. Also, we have constructed proofs only for a version of the MSX code in which the checksum loop in the `Continue_Build` procedure has been excised; the code is modified so that the test for the correct checksum is always passed. This has been done only because of lack of time to complete the work.

6.3 The INITIATE_BUILD adalemma

The `INITIATE_BUILD` procedure implements the first step in message processing. It examines the type of incoming messages, which (after elimination of a parity bit by the calling program `BUILD`) is contained in the first byte of the first command for a message. In several cases, all of the message is communicated by the first command, so `INITIATE_BUILD` places the message into the `App_Msg_Q` directly. Various error conditions are also handled by `INITIATE_BUILD`. Here, we concern ourselves only with the `Data_Struc_Load` messages, which do require more than one command and thus at least one call of `CONTINUE_BUILD`. The part of `INITIATE_BUILD` of interest is

```
procedure INITIATE_BUILD (Start_Op_Code : in GLOBAL_TYPES.BYTE;
```

[7] we would like to say "semantically equivalent," but Ada itself lacks the formal semantics that would be needed to formally establish semantic equivalence.

[8] However, the adalemma could only be used in the context of the modified version of BUILD, in which the "for" loop has been transformed to a procedure call. Eventually, enhancements to SDVS allowing adalemmas about arbitrary code fragments will obviate the need for this roundabout procedure.

```
                            Cmd_Buf_Index : in INTEGER) is
  Temp_word                   : GLOBAL_TYPES.WORD;
  High_Byte_Word              : GLOBAL_TYPES.WORD;
  Low_Byte_Word               : GLOBAL_TYPES.WORD;
  Data_Struc_ID_Found         : BOOLEAN;
  Start_Build                 : BOOLEAN;
  Struc_ID_Index              : INTEGER;
  X                           : INTEGER;
  Y                           : INTEGER;
begin
  Start_Build := False;
  case Start_Op_Code is
    when Data_Struc_Load =>
      Data_Struc_ID_Found := False;
      Struc_ID_Index :=
          CONV_BYTE_TO_INTEGER(Cmd_Buf(Cmd_Buf_Index).Third_Byte);
      if (Struc_ID_Index <= TRKING_DATA_STRUC.Num_Data_Struc) and
         (Struc_ID_Index > 0) then
        Cont_Cmd := Data_Load_Cmd;
        Num_Cmds_For_App_Msg  := Cmds_For_Data_Struc(Struc_ID_Index);
        Num_Words_For_App_Msg := TRKING_DATA_STRUC.Words_For_Data_Struc(
                                                      Struc_ID_Index) + 1;
        if (Num_Cmds_For_App_Msg /= 0) then
          Data_Struc_ID_Found := True;
          Start_Build := True;
        end if;
      end if;
      if (Data_Struc_ID_Found = False) then
        .
        .
        .
      end if;
    when RAM_Mem_Load | EEPROM_Mem_Load =>
        .
        .
        .
        .
        .
  end case;
  if Start_Build then
    App_Msg(1) := 0;
    App_Msg(2) := CONV_BYTE_TO_WORD(Start_Op_Code);
    Temp_word      := CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_buf_index).Second_Byte);
    High_Byte_Word := SHIFT_LOGICAL_WORD(Temp_word, 8);
    Temp_Word      := CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_buf_index).Third_Byte);
    App_Msg(3)     := OR_WORD(High_Byte_Word, Temp_Word);
    Temp_word      := CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_buf_index).Fourth_Byte);
    App_Msg(4)     := SHIFT_LOGICAL_WORD(Temp_word, 8);
    Starting_Cmd_With_Par := Cmd_Buf(Cmd_buf_index).First_Byte;
    Cmds_Rcvd := 1;
    Word_Cntr             := 4;
    Build_In_Progress     := True;
    Start_Build           := False;
  end if;
end INITIATE_BUILD;
```

What this code does is straightforward: After determining that the message is a Data_Struc_Load, it sets Cont_Cmd to Data_Load_Cmd, and determines the numbers of commands and words required for the incoming message, using look-up tables based on the message id—this is kept in Struc_ID_Index and is obtained from the third byte of the first command. After some validity checking, control passes to the if statement at the end of the procedure. Start_Build has been set to True by any part of the case statement processing a message that will involve additional commands, and thus must be "built." Through a sequence of conversions, bit-wise logical shifts, and logical-OR operations, the original four bytes of the command end up repackaged in words 2 through 4 of the App_Msg array. Word 1 of App_Msg is left zero; it is not involved in the processing of a Data_Struc_Load message and will ultimately be ignored by CONTINUE_BUILD.

We shall now discuss the corresponding adalemma for the Data_Struc_Load-message processing parts of INITIATE_BUILD. This adalemma and its proof, as saved by SDVS are:

```
(defadalemma initiate_build.data_struc_load.adalemma
    "msx_program_final_version_sans_checksum.a"      initiate_build
    msx_program_final_version_sans_checksum.app_msgs.initiate_build
    (".start_op_code = .data_struc_load"
     "~.build_in_progress"
     ".app_msgs.cmd_status_buf[.initiate_build.cmd_buf_index] = .cmd_ok"
     "is.byte(.start_op_code)"
     "is.cmd_bytes_type(.app_msgs.cmd_buf[.initiate_build.cmd_buf_index])"
     "origin(app_msgs.cmd_buf) le .initiate_build.cmd_buf_index"
     ".initiate_build.cmd_buf_index
       le (origin(app_msgs.cmd_buf) + range(app_msgs.cmd_buf)) - 1"
     "0 lt .record(app_msgs.cmd_buf[.initiate_build.cmd_buf_index],third_byte)"
     ".record(app_msgs.cmd_buf[.initiate_build.cmd_buf_index],third_byte)
       le .num_data_struc"
     ".cmds_for_data_struc
      [.record(app_msgs.cmd_buf[.initiate_build.cmd_buf_index],third_byte)] ~= 0")

     (cont_cmd
        num_cmds_for_app_msg
        num_words_for_app_msg
        (element app_msgs.cmd_status_buf (dot initiate_build.cmd_buf_index))
        (slice app_msg 1 4)
        starting_cmd_with_par
        cmds_rcvd
        word_cntr
        build_in_progress)

    ("forall q (origin(app_msg) le q & q le #word_cntr --> is.word(.app_msg[q]))"
     "#cont_cmd = .data_load_cmd"
     "#num_cmds_for_app_msg
       = .cmds_for_data_struc[.record(app_msgs.cmd_buf[.initiate_build.cmd_buf_index],
                                      third_byte)]"
     "#num_words_for_app_msg
       = .words_for_data_struc[.record(app_msgs.cmd_buf[.initiate_build.cmd_buf_index],
                                      third_byte)] + 1"
     "#build_in_progress = true"
     "#app_msg[1] = 0"
     "#app_msg[2] = .start_op_code"
    "high.byte(#app_msg[3],
             .record(app_msgs.cmd_buf[.initiate_build.cmd_buf_index], second_byte))"
    "low.byte(#app_msg[3],
            .record(app_msgs.cmd_buf[.initiate_build.cmd_buf_index],third_byte))"
    "high.byte(#app_msg[4],
             .record(app_msgs.cmd_buf[.initiate_build.cmd_buf_index], fourth_byte))"
    "low.byte(#app_msg[4],0)"
    "#starting_cmd_with_par
      = .record(app_msgs.cmd_buf[.initiate_build.cmd_buf_index],first_byte)"
    "#cmds_rcvd = 1"
    "#word_cntr = 4")

  :proof
"(proveadalemma initiate_build.data_struc_load.adalemma
     proof:
(readaxioms \"/u/versys/sdvs/axioms/arraycoverings.axioms\",
 provebyaxiom pcovering(app_msg[1:4],app_msg[1])
   using: pcovering\slice\element,
 provebyaxiom pcovering(app_msg[1:4],app_msg[2])
   using: pcovering\slice\element,
 provebyaxiom pcovering(app_msg[1:4],app_msg[3])
   using: pcovering\slice\element,
 provebyaxiom pcovering(app_msg[1:4],app_msg[4])
   using: pcovering\slice\element,
 applydecls,
 go
  #msx_program_final_version_sans_checksum\pc
     = at(msx_program_final_version_sans_checksum.intrinsic_functions.shift_logical_word),
 invokeadalemma shift_logical_word.adalemma,
 go
  #msx_program_final_version_sans_checksum\pc
     = at(msx_program_final_version_sans_checksum.intrinsic_functions.or_word),
 invokeadalemma or_word.adalemma,
 go
  #msx_program_final_version_sans_checksum\pc
     = at(msx_program_final_version_sans_checksum.intrinsic_functions.shift_logical_word),
 invokeadalemma shift_logical_word.adalemma,
 go
  #msx_program_final_version_sans_checksum\pc
     = exited(msx_program_final_version_sans_checksum.app_msgs.initiate_build),
 notice forall q (q le 4 --> q = 4 or q le 3),
 notice forall q (q le 3 --> q = 3 or q le 2),
 notice forall q (q le 2 --> q = 2 or q le 1),
 notice forall q is.word(.app_msg[1]),
 notice forall q is.word(.app_msg[2]),
 notice forall q is.word(.app_msg[3]),
 notice forall q is.word(.app_msg[4]),
 provebygeneralization g(1)
   using: (q(1),q(2),q(3),q(4),q(5),q(6),q(7)),
 close))")
```

We shall discuss the various parts of the adalemma and its proof. The first group of quoted lines are the precondition, those facts which must be true in order that the adalemma may be applied. In general, these assertions reconstruct the environment within which the target procedure is called. For example, INITIATE_BUILD is called only when Build_In_Progress is False, so the precondition asserts that fact. When SDVS sets up the environment for proving the adalemma, the preconditions are asserted (as in the proof of any state delta).

The Precondition:

```
".start_op_code = .data_struc_load"
"~.build_in_progress"
".app_msgs.cmd_status_buf[.initiate_build.cmd_buf_index] = .cmd_ok"
"is.byte(.start_op_code)"
"is.cmd_bytes_type(.app_msgs.cmd_buf[.initiate_build.cmd_buf_index])"
"origin(app_msgs.cmd_buf) le .initiate_build.cmd_buf_index"
".initiate_build.cmd_buf_index le
   (origin(app_msgs.cmd_buf) + range(app_msgs.cmd_buf)) - 1"
"0 lt .record(app_msgs.cmd_buf[.initiate_build.cmd_buf_index],third_byte)"
".record(app_msgs.cmd_buf[.initiate_build.cmd_buf_index],third_byte) le
                  .num_data_struc"
".cmds_for_data_struc[
   .record(app_msgs.cmd_buf[.initiate_build.cmd_buf_index],third_byte)] ~= 0")
```

All of the assertions are either facts which are true in the environment in which INITIATE_BUILD is called, or which restrict applicability to the case of interest (i.e., the Data_Struc_Load messages), and which are actually necessary for the proof. The last three of the preconditions, referring to the third byte of the command for which INITIATE_BUILD is called, assert the legitimacy of this value as a message id, and that the corresponding message length is not 0.

The following mod list is a list of places which are altered by INITIATE_BUILD.

The Mod List:

```
cont_cmd,
num_cmds_for_app_msg,
num_words_for_app_msg,
app_msgs.cmd_status_buf[.initiate_build.cmd_buf_index],
app_msg[1:4],
starting_cmd_with_par,
cmds_rcvd,
word_cntr,
build_in_progress
```

The proof for the INITIATE_BUILD adalemma is short enough that we can discuss it at this point.

The Proof:

```
proveadalemma initiate_build.data_struc_load.adalemma
      proof:
(readaxioms \"/u/versys/sdvs/axioms/arraycoverings.axioms\",
 provebyaxiom pcovering(app_msg[1:4],app_msg[1])
```

```
  using: pcovering\slice\element,
provebyaxiom pcovering(app_msg[1:4],app_msg[2])
  using: pcovering\slice\element,
provebyaxiom pcovering(app_msg[1:4],app_msg[3])
  using: pcovering\slice\element,
provebyaxiom pcovering(app_msg[1:4],app_msg[4])
  using: pcovering\slice\element,
applydecls,
go
 #msx_program_final_version_sans_checksum\pc
    = at(msx_program_final_version_sans_checksum.intrinsic_functions.shift_logical_word),
invokeadalemma shift_logical_word.adalemma,
go
 #msx_program_final_version_sans_checksum\pc
    = at(msx_program_final_version_sans_checksum.intrinsic_functions.or_word),
invokeadalemma or_word.adalemma,
go
 #msx_program_final_version_sans_checksum\pc
    = at(msx_program_final_version_sans_checksum.intrinsic_functions.shift_logical_word),
invokeadalemma shift_logical_word.adalemma,
go
 #msx_program_final_version_sans_checksum\pc
    = exited(msx_program_final_version_sans_checksum.app_msgs.initiate_build),
notice forall q (q le 4 --> q = 4 or q le 3),
notice forall q (q le 3 --> q = 3 or q le 2),
notice forall q (q le 2 --> q = 2 or q le 1),
notice forall q is.word(.app_msg[1]),
notice forall q is.word(.app_msg[2]),
notice forall q is.word(.app_msg[3]),
notice forall q is.word(.app_msg[4]),
provebygeneralization g(1)
  using: (q(1),q(2),q(3),q(4),q(5),q(6),q(7)),
close)
```

The provebyaxiom commands at the beginning establish that array places actually changed during symbolic execution are covered by the mod list of the adalemma. Each call to an intrinsic function is handled by an adalemma. The series of notices at the end are connected with a familiar problem: the simplifier automatically has available that $n \leq m \rightarrow n = m \vee n \leq m - 1$, but must be guided to extend this further to such facts as $n \leq m \rightarrow n = m \vee n = m - 1 \vee n = m - 2 \vee n \leq m - 3$.

6.4 The CONTINUE_BUILD adalemmas

6.4.1 Overview

The CONTINUE_BUILD procedure is called to handle continuation commands, i.e., commands containing parts of a message for which a build is in progress. Since we are considering only Data_Struc_Load messages, we will consider only the handling of Data_Load_Cmds by CONTINUE_BUILD; Data_Load_Cmd is the value set in Cont_Cmd by INITIATE_BUILD when it starts the build of a Data_Struc_Load message. The parts of CONTINUE_BUILD of interest to us are the following:

```
procedure CONTINUE_BUILD(Cont_Op_Code  : in GLOBAL_TYPES.BYTE;
           Cmd_Buf_Index : in INTEGER) is
  App_Msg_Checksum      : GLOBAL_TYPES.WORD;
  App_Msg_Start_Word    : INTEGER;
  Msg_Word_Index        : INTEGER;
  Q_Word_Index          : INTEGER;
  Temp_Word             : GLOBAL_TYPES.WORD;
  High_Byte_Word        : GLOBAL_TYPES.WORD;
  Low_Byte_Word         : GLOBAL_TYPES.WORD;
begin
  if Cont_Op_Code = Cont_Cmd then
    if (Cmds_Rcvd mod 2) = 0 then
      Word_Cntr := Word_Cntr + 1;
      Temp_Word      := CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_Buf_Index).Second_Byte);
      High_Byte_Word := SHIFT_LOGICAL_WORD(Temp_Word, 8);
      Low_Byte_Word  := CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_Buf_Index).Third_Byte);
      App_Msg(Word_Cntr) := OR_WORD(High_Byte_Word, Low_Byte_Word);
      Word_Cntr := Word_Cntr + 1;
      Temp_Word := CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_Buf_Index).Fourth_Byte);
      App_Msg(Word_Cntr) := SHIFT_LOGICAL_WORD(Temp_Word, 8);
    else
      Low_Byte_Word  := CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_Buf_Index).Second_Byte);
      App_Msg(Word_Cntr) := OR_WORD(App_Msg(Word_Cntr), Low_Byte_Word);
      Word_Cntr := Word_Cntr + 1;
      Temp_Word      := CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_Buf_Index).Third_Byte);
      High_Byte_Word := SHIFT_LOGICAL_WORD(Temp_Word, 8);
      Low_Byte_Word  := CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_Buf_Index).Fourth_Byte);
      App_Msg(Word_Cntr) := OR_WORD(High_Byte_Word, Low_Byte_Word);
    end if;
    Cmds_Rcvd := Cmds_Rcvd + 1;
    if Cmds_Rcvd = Num_Cmds_For_App_Msg then
      App_Msg_Checksum := 0;
      App_Msg_Checksum := App_Msg(Num_Words_For_App_Msg + 1);
      if App_Msg_Checksum = App_Msg(Num_Words_For_App_Msg + 1) then
        if App_Msg_Counter < App_Msg_Q_Size then
          App_Msg_Start_Word := 2;
          if (CONV_WORD_TO_BYTE(App_Msg(2)) = RAM_Mem_Load) or
             (CONV_WORD_TO_BYTE(App_Msg(2)) = EEPROM_Mem_Load) or
             (CONV_WORD_TO_BYTE(App_Msg(2)) = Mem_Dump_X_Frames) then
                .
                .
                .
                .
          end if;
          Q_Tail := (Q_Tail mod App_Msg_Q_Size) + 1;
          Num_Words_For_App_Msg := Num_Words_For_App_Msg + 1;
          Q_Word_Index := 0;
          for Msg_Word_Index in App_Msg_Start_Word..Num_Words_For_App_Msg loop
            Q_Word_Index := Q_Word_Index + 1;
            App_Msg_Q(Q_Tail)(Q_Word_Index) := App_Msg(Msg_Word_Index);
          end loop;
               App_Msg_Q(Q_Tail)(CMDS_TYPES.App_Msg_Max) := Q_Word_Index;
          App_Msg_Counter := App_Msg_Counter + 1;
          if App_Msg_Start_Word = 1 then
                null;
          else
            Temp_Word := SHIFT_LOGICAL_WORD(App_Msg(3), -8);
          end if;
          Temp_Word := SHIFT_LOGICAL_WORD(App_Msg(4), -8);
        else
                .
                .
                .
                .
        end if;
      else
        .
        .
        .
        .
      end if;
      Build_In_Progress := False;
    end if;
  else
        .
        .
        .
        .
  end if;
end CONTINUE_BUILD;
```

There are two adalemmas for CONTINUE_BUILD; one handles the case where the command to be processed is not the last command for the message being built, and the other is for the case where the current command is the last one. When the last command is processed, CONTINUE_BUILD copies the completed message into the App_Msg_Q. It is true that the two adalemmas could be combined, because all the functions performed in the "not last" case are also performed in the "last" case; one would add the additional facts to the postcondition in the form of conditional assertions based on whether #Cmds_Rcvd = .Num_Cmds_For_App_Msg. However, this complicates the exposition somewhat, and we prefer to use separate adalemmas. We will first discuss the adalemma for the "not last" case.

6.4.2 The "not last" command case

```
(defadalemma continue_build.data_load_cmd.not.last.adalemma
    "msx_program_final_version_sans_checksum.a"
    continue_build
    msx_program_final_version_sans_checksum.app_msgs.continue_build
    (".build_in_progress"
     ".app_msgs.cmd_status_buf[.continue_build.cmd_buf_index] = .cmd_ok"
     "is.byte(.starting_cmd_with_par)"
     "is.byte(.cont_cmd)"
     "is.byte(.cont_op_code)"
     ".cont_cmd = .data_load_cmd"
     ".cont_op_code = .cont_cmd"
```

```
  "1 le .cmds_rcvd"
  ".cmds_rcvd lt .num_cmds_for_app_msg - 1"
  ".cmds_rcvd mod 2 = .cmds_rcvd - (.cmds_rcvd / 2) * 2"
  ".cmds_rcvd mod 2 = 0 or .cmds_rcvd mod 2 = 1"

  ".cmds_rcvd mod 2 = 1 --> low.byte(.app_msg[.word_cntr],0)"

  "4 le .word_cntr"
  ".word_cntr le 78"
  "is.cmd_bytes_type(.app_msgs.cmd_buf[.continue_build.cmd_buf_index])"
  "origin(app_msgs.cmd_buf) le .continue_build.cmd_buf_index"
  ".continue_build.cmd_buf_index
    le (origin(app_msgs.cmd_buf) + range(app_msgs.cmd_buf)) - 1"

  "forall q (origin(app_msg) le q & q le .word_cntr --> is.word(.app_msg[q]))"
 )

(word_cntr
  (element app_msg (minus (plus (dot word_cntr) 1) (mod (dot cmds_rcvd) 2)))
  (element app_msg (minus (plus (dot word_cntr) 2) (mod (dot cmds_rcvd) 2)))
  cmds_rcvd)

 ("forall q (origin(app_msg) le q & q le #word_cntr --> is.word(.app_msg[q]))"
  "#word_cntr = (.word_cntr + 2) - .cmds_rcvd mod 2"
  "#cmds_rcvd = .cmds_rcvd + 1"
  ".cmds_rcvd mod 2 = 0 --> #word_cntr = .word_cntr + 2"
  ".cmds_rcvd mod 2 = 0
    --> high.byte(#app_msg[.word_cntr + 1],
                  .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                          second_byte))"
  ".cmds_rcvd mod 2 = 0
    --> low.byte(#app_msg[.word_cntr + 1],
                 .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                         third_byte))"
  ".cmds_rcvd mod 2 = 0
    --> high.byte(#app_msg[.word_cntr + 2],
                  .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                          fourth_byte))"
  ".cmds_rcvd mod 2 = 0 --> low.byte(#app_msg[.word_cntr + 2],0)"

  ".cmds_rcvd mod 2 = 1 --> #word_cntr = .word_cntr + 1"
  ".cmds_rcvd mod 2 = 1
    --> high.byte(#app_msg[.word_cntr],.app_msg[.word_cntr] / 256)"
  ".cmds_rcvd mod 2 = 1
    --> low.byte(#app_msg[.word_cntr],
                 .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                         second_byte))"
  ".cmds_rcvd mod 2 = 1
    --> high.byte(#app_msg[.word_cntr + 1],
                  .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                          third_byte))"
  ".cmds_rcvd mod 2 = 1
    --> low.byte(#app_msg[.word_cntr + 1],
                 .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                         fourth_byte))"))
```

We shall discuss the various parts of this adalemma.

First consider the following precondition:

```
".build_in_progress"
".app_msgs.cmd_status_buf[.continue_build.cmd_buf_index] = .cmd_ok"
"is.byte(.starting_cmd_with_par)"
"is.byte(.cont_cmd)"
"is.byte(.cont_op_code)"
".cont_cmd = .data_load_cmd"
".cont_op_code = .cont_cmd"
"1 le .cmds_rcvd"
".cmds_rcvd lt .num_cmds_for_app_msg - 1"

".cmds_rcvd mod 2 = .cmds_rcvd - (.cmds_rcvd / 2) * 2"
".cmds_rcvd mod 2 = 0 or .cmds_rcvd mod 2 = 1"

".cmds_rcvd mod 2 = 1 --> low.byte(.app_msg[.word_cntr],0)"

"4 le .word_cntr"
".word_cntr le 78"
"is.cmd_bytes_type(.app_msgs.cmd_buf[.continue_build.cmd_buf_index])"
"origin(app_msgs.cmd_buf) le .continue_build.cmd_buf_index"
".continue_build.cmd_buf_index
  le (origin(app_msgs.cmd_buf) + range(app_msgs.cmd_buf)) - 1"

"forall q (origin(app_msg) le q & q le .word_cntr --> is.word(.app_msg[q]))"
```

The first group of assertions set the environment within which `CONTINUE_BUILD` might be called, and the circumstances under which this adalemma is to be applied. There are two assertions giving properties of the `mod` function—of course, these are mathematically provable facts, but they are not automatically known to the simplifier, so we have put them into the precondition of the adalemma. The user will be responsible for proving these before the adalemma is applied. This is important, because the expression `.cmds_rcvd mod 2` is used in the mod list of this adalemma, and the correct covering properties will not be known to the covering solver unless the possible values of this expression are known. The assertion about `.app_msg[.word_cntr]` is that if an odd number of commands have been received, then the low byte of the last word in the current message is 0. The next group of assertions guarantee that various array indices are in range. The last assertion states that everything in the `.app_msg` array is a "word," i.e., is an integer in the range 0 ... 65535. SDVS has the built-in type *integer* but the simplifier does not handle subtypes; to deal with this situation, the SDVS Ada translator places guard conditions on its translations of assignment statements for variables declared of a given subtype, to insure that the value assigned is actually in the subtype. If a variable `V` is declared as a word, then the translator output requires that any quantities to be assigned to `V` must be in the range 0 ... 65535. Since members of the `App_Msg` array are later used in assignments to other words, the information that `App_Msg` consists of words must be available. Indeed, there is such a step even within the code for `CONTINUE_BUILD`. Hence, the quantified statement must be part of the precondition to `CONTINUE_BUILD`, not merely proved after its invocation. A similar statement, with an appropriately enlarged range of subscripts for .app_msg, is part of the postcondition.

Next consider the following mod list (in input notation):

```
word_cntr,
app_msg[.word_cntr+1-(.cmds_rcvd mod 2)],
app_msg[.word_cntr+2-(.cmds_rcvd mod 2)],
cmds_rcvd
```

These are simply a list of the places which are modified by CONTINUE_BUILD when called on a command which is not a "last" command. As it happens, we were able to write expressions for the array subscripts which work out correctly whether or not .cmds_rcvd is odd or even—were this not so, we should have been obliged to prove two separate adalemmas, one for each of these cases.

The postcondition is examined in several parts:

```
"forall q (origin(app_msg) le q & q le #word_cntr --> is.word(.app_msg[q]))"
     "#word_cntr = (.word_cntr + 2) - .cmds_rcvd mod 2"
     "#cmds_rcvd = .cmds_rcvd + 1"
     ".cmds_rcvd mod 2 = 0 --> #word_cntr = .word_cntr + 2"
```

The quantified statement above asserts, as promised, that app_msg is an array of words. Then the new values of word_cntr and cmds_rcvd are asserted.

```
     ".cmds_rcvd mod 2 = 0
       --> high.byte(#app_msg[.word_cntr + 1],
                     .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                             second_byte))"
     ".cmds_rcvd mod 2 = 0
       --> low.byte(#app_msg[.word_cntr + 1],
                    .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                            third_byte))"
     ".cmds_rcvd mod 2 = 0
       --> high.byte(#app_msg[.word_cntr + 2],
                     .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                             fourth_byte))"
     ".cmds_rcvd mod 2 = 0 --> low.byte(#app_msg[.word_cntr + 2],0)"
```

This group of assertions applies when .cmds_rcvd is initially even. In that case, the three bytes of the current command are stored into the two bytes of the next available word in app_msg and the high byte of the word after that.

It is appropriate at this point to look at the definitions of the macros high.byte and low.byte. They are

```
(defmacroo high.byte        ;extra facts here because this is used in postconditions
    "0 le x & x le 65535 & y = x / 256 & 0 le y & y le 255 &
```

```
              x = y * 256 + x mod 256"
  (x y) nil)

(defmacroo low.byte        ;extra facts here because this is used in postconditions
    "0 le x & x le 65535 & y = x mod 256 & 0 le y & y le 255 &
              x = ((x / 256) * 256) + y"
  (x y) nil)
```

The reader will note that these seem redundant; some of what is given is a mathematical consequence of the rest. However, if we left it at that, we should have to prove those consequences at some point; putting them instead in these macros provides the simplifier more information to work with, and eliminates (we hope) the need for the user to reprove these things.

The last group of assertions is for the case that the initial value of `.cmds_rcvd` is an odd integer:

```
    ".cmds_rcvd mod 2 = 1 --> #word_cntr = .word_cntr + 1"
    ".cmds_rcvd mod 2 = 1
      --> high.byte(#app_msg[.word_cntr],.app_msg[.word_cntr] / 256)"
    ".cmds_rcvd mod 2 = 1
      --> low.byte(#app_msg[.word_cntr],
                  .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                          second_byte))"
    ".cmds_rcvd mod 2 = 1
      --> high.byte(#app_msg[.word_cntr + 1],
                   .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                           third_byte))"
    ".cmds_rcvd mod 2 = 1
      --> low.byte(#app_msg[.word_cntr + 1],
                  .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                          fourth_byte))"))
```

This group applies when the number of previously received commands is odd. In such a case, the high byte of the last used word in `App_Msg` is preserved, and the three bytes from the current command are inserted into the low byte of that word, and the two bytes of the next available word in `App_Msg`.

This `CONTINUE_BUILD` adalemma would normally be applied in a situation in which there is a quantified statement expressing the relationship between the `Cmd_Buf` and the current contents of `App_Msg`. The lemma must be constructed so that its application does not cause the deletion of that statement from the usablequantifiers list, and so that an appropriately enlarged quantified statement can be proved after the application of the adalemma. These requirements translate into the need for an accurate mod list, and a complete rendering of the new situation in the postcondition.

6.4.3 The "last" command case

The second adalemma for `CONTINUE_BUILD` addresses the situation that the current command is in fact the last one required for the build of the message to be complete: this occurs when

```
        .cmds_rcvd = .num_cmds_for_app_msg - 1
```

In this case, not only does CONTINUE_BUILD perform the same steps as before, but it also copies the completed message into the App_Msg_Q. Therefore there are correspondingly more places listed in the mod list and more assertions in the postcondition.[9]

The adalemma for the "last" command is

```
(defadalemma continue_build.data_load_cmd.last.not.eeprom.adalemma
    "msx_program_final_version_sans_checksum.a"
    continue_build
    msx_program_final_version_sans_checksum.app_msgs.continue_build
    (".build_in_progress"
     ".app_msgs.cmd_status_buf[.continue_build.cmd_buf_index] = .cmd_ok"
     "is.byte(.starting_cmd_with_par)"
     "is.byte(.cont_cmd)"
     "is.byte(.cont_op_code)"
     ".cont_cmd = .data_load_cmd"
     ".cont_op_code = .cont_cmd"
     "1 le .cmds_rcvd"
     ".cmds_rcvd = .num_cmds_for_app_msg - 1"
     ".cmds_rcvd mod 2 = .cmds_rcvd - (.cmds_rcvd / 2) * 2"
     ".cmds_rcvd mod 2 = 0 or .cmds_rcvd mod 2 = 1"
     ".cmds_rcvd mod 2 = 1 --> low.byte(.app_msg[.word_cntr],0)"
     "4 le .word_cntr"
     ".word_cntr le 78"
     "is.cmd_bytes_type(.app_msgs.cmd_buf[.continue_build.cmd_buf_index])"
     "origin(app_msgs.cmd_buf) le .continue_build.cmd_buf_index"
     ".continue_build.cmd_buf_index
       le (origin(app_msgs.cmd_buf) + range(app_msgs.cmd_buf)) - 1"

     "3 le .num_words_for_app_msg"
     ".num_words_for_app_msg le (range(app_msg) + origin(app_msg)) - 4"

     "forall q (origin(app_msg) le q & q le .word_cntr --> is.word(.app_msg[q]))"

     "(.word_cntr + 2) - .cmds_rcvd mod 2 ge .num_words_for_app_msg + 1"

     ".app_msg_counter lt .app_msg_q_size"

     "0 le .q_tail"
     ".q_tail = 0 --> .app_msg_counter = 0"
     ".q_tail le (range(app_msg_q) + origin(app_msg_q)) - 1"
     ".q_tail lt range(app_msg_q) & .q_tail mod .app_msg_q_size = .q_tail or
      .q_tail = range(app_msg_q) & .q_tail mod .app_msg_q_size = 0"
     "30 mod 30 = 0"
     "is.byte(.app_msg[2])"
```

[9] Actually, there is a further case distinction made in CONTINUE_BUILD: whether the message completed is a RAM_Mem_Load, a EEPROM_Mem_Load, or a Mem_Dump_X_Frames message; in any of these cases further processing of the message is performed before it is placed in the App_Msg_Q. But we are restricting consideration in this report to the Data_Struc_Load messages, which do not require any further processing. Nevertheless, we have proved the adalemma for even the more complex situation of the other message types mentioned. This third CONTINUE_BUILD adalemma will be listed at the end of this section, but will not be discussed further.

```
 ".app_msg[2] ~= .ram_mem_load"
 ".app_msg[2] ~= .eeprom_mem_load"
 ".app_msg[2] ~= .mem_dump_x_frames")
 (word_cntr
  (element app_msg (minus (plus (dot word_cntr) 1) (mod (dot cmds_rcvd) 2)))
  (element app_msg (minus (plus (dot word_cntr) 2) (mod (dot cmds_rcvd) 2)))
  cmds_rcvd
  q_tail
  num_words_for_app_msg
  (slice (element app_msg_q (plus (mod (dot q_tail) (dot app_msg_q_size)) 1))
    1
    (plus (dot num_words_for_app_msg) 1))
  (element (element app_msg_q (plus (mod (dot q_tail) (dot app_msg_q_size)) 1))
    (dot app_msg_max))
  app_msg_counter
  cmd_last_app_msg
  build_in_progress)
("forall q (origin(app_msg) le q & q le #word_cntr --> is.word(.app_msg[q]))"
 "forall q (1 le q & q le .num_words_for_app_msg
           --> #app_msg_q[.q_tail mod .app_msg_q_size + 1][q]
                 = #app_msg[q + 1])"
 "#word_cntr = (.word_cntr + 2) - .cmds_rcvd mod 2"
 "#cmds_rcvd = .cmds_rcvd + 1"
 ".cmds_rcvd mod 2 = 0 --> #word_cntr = .word_cntr + 2"
 ".cmds_rcvd mod 2 = 0
   --> high.byte(#app_msg[.word_cntr + 1],
                 .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                         second_byte))"
 ".cmds_rcvd mod 2 = 0
   --> low.byte(#app_msg[.word_cntr + 1],
                .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                        third_byte))"
 ".cmds_rcvd mod 2 = 0
   --> high.byte(#app_msg[.word_cntr + 2],
                 .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                         fourth_byte))"
 ".cmds_rcvd mod 2 = 0 --> low.byte(#app_msg[.word_cntr + 2],0)"
 ".cmds_rcvd mod 2 = 1 --> #word_cntr = .word_cntr + 1"
 ".cmds_rcvd mod 2 = 1
   --> high.byte(#app_msg[.word_cntr],.app_msg[.word_cntr] / 256)"
 ".cmds_rcvd mod 2 = 1
   --> low.byte(#app_msg[.word_cntr],
                .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                        second_byte))"
 ".cmds_rcvd mod 2 = 1
   --> high.byte(#app_msg[.word_cntr + 1],
                 .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                         third_byte))"
 ".cmds_rcvd mod 2 = 1
   --> low.byte(#app_msg[.word_cntr + 1],
                .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                        fourth_byte))"
 "#q_tail = .q_tail mod .app_msg_q_size + 1"
 "origin(app_msg_q) le #q_tail &
  #q_tail le (range(app_msg_q) + origin(app_msg_q)) - 1"
```

```
    "#num_words_for_app_msg = .num_words_for_app_msg + 1"
    "#app_msg_q[.q_tail mod .app_msg_q_size + 1][.app_msg_max]
      = #num_words_for_app_msg - 1"
    "#app_msg_counter = .app_msg_counter + 1"
    "#build_in_progress = false"))
```

The precondition, mod list, and postcondition of this adalemma are each an extension of the corresponding part of the first adalemma. We shall discuss only the differences.

In the precondition, there are assertions concerning `.num_words_for_app_msg`, which express what is known about this quantity at times when `CONTINUE_BUILD` is invoked. There are several assertions about `.q_tail`, which serve to place it in the correct range, and also to guide the simplifier in determining bounds on the new value, `#q_tail`. This includes arithmetical facts about the result of using the `mod` function, which are mathematically true but not automatically available to the simplifier. The initial value for `.q_tail` is 0, but once anything has been written into the `App_Msg_Q`, `Q_tail` is never again 0. Thus, `.q_tail = 0 --> .app_msg_counter = 0` is true, although this fact is never used anywhere in the proof. (Note that the converse does not hold: `.app_msg_q = 0` is possible even when `.q_tail ~= 0`.) The "`30 mod 30 = 0`" is present merely to coax the simplifier in the right direction; this fact is automatically known if the simplifier is interrogated about it, but apparently the act of interrogation, or including it in the precondition, increases its availability for other behind-the-scenes work of the simplifier. Inserting this arithmetical fact in the precondition saves a few "notice" commands later on in the proof. `App_Msg(2)` contains the message id in its low-order byte. The exact value of `.app_msg[2]` would likely be known to the simplifier at the moment of any particular application of this adalemma, but we are stating and proving the adalemma in a general situation without any knowledge of an exact value. Thus, we have precondition assertions giving needed facts about `.app_msg[2]`.[10]

The mod list is, in user-interface instead of lisp notation,

```
word_cntr,
app_msg[.word_cntr+1-(.cmds_rcvd mod 2)],
app_msg[.word_cntr+2-(.cmds_rcvd mod 2)],
cmds_rcvd,
q_tail,
num_words_for_app_msg,
app_msg_q[.q_tail mod .app_msg_q_size + 1][1:.num_words_for_app_msg + 1],
app_msg_q[.q_tail mod .app_msg_q_size + 1][.app_msg_max],
app_msg_counter,
build_in_progress
```

The additional places are simply those that may be altered when the completed message is copied into the `App_Msg_Q`.

The additional parts of the postcondition are mostly self-explanatory, merely expressing what is accomplished by the additional code executed in the case of a completed message. Noteworthy are the assertions concerning `.app_msg_q`:

[10]In particular, the assertions that the current message is not one of `RAM_Mem_Load`, `EEPROM_Mem_Load`, or `Mem_Dump_X_Frames`.

```
"forall q (1 le q & q le .num_words_for_app_msg
    --> #app_msg_q[.q_tail mod .app_msg_q_size + 1][q] = #app_msg[q + 1])"
"#app_msg_q[.q_tail mod .app_msg_q_size + 1][.app_msg_max]
    = #num_words_for_app_msg - 1"
```

These indicate that the completed message has been copied to the `App_Msg_Q`.

The two adalemmas we have given suffice for handling the calls to `CONTINUE_BUILD` when the message being built is a `Data_Struc_Load`. We also proved an adalemma for the remaining cases of the `Data_Load_Cmd`, namely that it is one of `RAM_Mem_Load`, `EEPROM_Mem_Load`, or `Mem_Dump_X_Frames`. This we present here, but we shall not discuss it further in this report:

```
(defadalemma continue_build.data_load_cmd.last.eeprom.adalemma
    "msx_program_final_version_sans_checksum.a"
     continue_build
     msx_program_final_version_sans_checksum.app_msgs.continue_build
    (".build_in_progress"
     ".app_msgs.cmd_status_buf[.continue_build.cmd_buf_index] = .cmd_ok"
     "is.byte(.starting_cmd_with_par)"
     "is.byte(.cont_cmd)"
     "is.byte(.cont_op_code)"
     ".cont_cmd = .data_load_cmd"
     ".cont_op_code = .cont_cmd"
     "1 le .cmds_rcvd"
     ".cmds_rcvd = .num_cmds_for_app_msg - 1"
     ".cmds_rcvd mod 2 = .cmds_rcvd - (.cmds_rcvd / 2) * 2"
     ".cmds_rcvd mod 2 = 0 or .cmds_rcvd mod 2 = 1"
     ".cmds_rcvd mod 2 = 1 --> low.byte(.app_msg[.word_cntr],0)"
     "4 le .word_cntr"
     ".word_cntr le 78"
     "is.cmd_bytes_type(.app_msgs.cmd_buf[.continue_build.cmd_buf_index])"
     "origin(app_msgs.cmd_buf) le .continue_build.cmd_buf_index"
     ".continue_build.cmd_buf_index
       le (origin(app_msgs.cmd_buf) + range(app_msgs.cmd_buf)) - 1"
     "3 le .num_words_for_app_msg"
     ".num_words_for_app_msg
       le ((range(app_msg) + origin(app_msg)) - 1) - 3"
     "forall q (origin(app_msg) le q & q le .word_cntr --> is.word(.app_msg[q]))"
     "(.word_cntr + 2) - .cmds_rcvd mod 2 ge .num_words_for_app_msg + 1"
     ".app_msg_counter lt .app_msg_q_size"
     "0 le .q_tail"
     ".q_tail = 0 --> .app_msg_counter = 0"
     ".q_tail le (range(app_msg_q) + origin(app_msg_q)) - 1"
     ".q_tail lt range(app_msg_q) &
       .q_tail mod .app_msg_q_size + 1 = .q_tail + 1 or
      .q_tail = range(app_msg_q) &
        .q_tail mod .app_msg_q_size + 1 = 1"
     "30 mod 30 = 0"
     ".app_msg[2] = .eeprom_mem_load")
(word_cntr
(element app_msg (minus (plus (dot word_cntr) 1) (mod (dot cmds_rcvd)
2)))
(element app_msg (minus (plus (dot word_cntr) 2) (mod (dot cmds_rcvd)
2)))
```

```
cmds_rcvd
(element app_msg 1)
(element app_msg 2)
(element app_msg 3)
q_tail
num_words_for_app_msg
(slice (element app_msg_q (plus (mod (dot q_tail) (dot app_msg_q_size))
   1)) 1 (plus (dot num_words_for_app_msg) 1))
(element (element app_msg_q (plus (mod (dot q_tail) (dot app_msg_q_size)) 1))
   (dot app_msg_max))
app_msg_counter
     cmd_last_app_msg
build_in_progress)
    ("forall q (origin(app_msg) le q & q le #word_cntr --> is.word(.app_msg[q]))"
     "forall q (1 le q & q le .num_words_for_app_msg
                --> #app_msg_q[.q_tail mod .app_msg_q_size + 1][q] = #app_msg
                                                                     [q])"
     "#word_cntr = (.word_cntr + 2) - .cmds_rcvd mod 2"
     "#cmds_rcvd = .cmds_rcvd + 1"
     ".cmds_rcvd mod 2 = 0 --> #word_cntr = .word_cntr + 2"
     ".cmds_rcvd mod 2 = 0
       --> high.byte(#app_msg[.word_cntr + 1],
                    .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                            second_byte))"
     ".cmds_rcvd mod 2 = 0
       --> low.byte(#app_msg[.word_cntr + 1],
                   .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                           third_byte))"
     ".cmds_rcvd mod 2 = 0
       --> high.byte(#app_msg[.word_cntr + 2],
                    .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                            fourth_byte))"
     ".cmds_rcvd mod 2 = 0 --> low.byte(#app_msg[.word_cntr + 2],0)"
     ".cmds_rcvd mod 2 = 1 --> #word_cntr = .word_cntr + 1"
     ".cmds_rcvd mod 2 = 1
       --> high.byte(#app_msg[.word_cntr],.app_msg[.word_cntr] / 256)"
     ".cmds_rcvd mod 2 = 1
       --> low.byte(#app_msg[.word_cntr],
                   .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                           second_byte))"
     ".cmds_rcvd mod 2 = 1
       --> high.byte(#app_msg[.word_cntr + 1],
                    .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                            third_byte))"
     ".cmds_rcvd mod 2 = 1
       --> low.byte(#app_msg[.word_cntr + 1],
                   .record(app_msgs.cmd_buf[.continue_build.cmd_buf_index],
                           fourth_byte))"
     "#app_msg[1] = .app_msg[2]"
     "#app_msg[2] = .app_msg[3] / 256"
     "#app_msg[3] = .app_msg[3] mod 256"
     "#q_tail = .q_tail mod .app_msg_q_size + 1"
     "origin(app_msg_q) le #q_tail &
      #q_tail le (range(app_msg_q) + origin(app_msg_q)) - 1"
     "#num_words_for_app_msg = .num_words_for_app_msg + 1"
```

```
"#app_msg_q[.q_tail mod .app_msg_q_size + 1][.app_msg_max]
  = #num_words_for_app_msg"
"#app_msg_counter = .app_msg_counter + 1"
"#build_in_progress = false"))
```

It should be noted that the proofs for the adalemmas in the "last" command cases do not close properly. They fail during the final steps of undeclaration of local variables, at a point where all goals apart from the "exited" condition have been proved. We have determined that the problem is due to a bug in SDVS, in which a wrong choice is made for a symbol to be used for a subprogram variable. The bug is possibly some type of "off by one" error. Since all other goals are proved before the bug manifests, we expect that the same proofs we have developed will close properly when the bug is fixed.

7 Conclusions

The primary purpose of this verification effort was to stress-test SDVS and help us formulate further research and development strategies. This project lasted only a year, and because (1) this was the first large Ada application verified and (2) SDVS development occurred concurrently with the program verification, we were restricted in the complexity of the specification and amount of code that could be tackled. However, we found this experience very important for identifying weak and missing features in the tools and underlying theory, and it is helping us prioritize future research and development tasks (see [6] for details).

This application has confirmed our belief that it is not always clear how to scale up (i.e., to verify large applications) from experiences with small examples. The Ada program we verified was by far the largest Ada program to be translated by the SDVS Ada translator, and we ran into unexpected storage and time problems. Perhaps some of these problems are unavoidable given the size of the program. But it was only towards the end of the project that we took a more promising approach to the verification of the program: proving adalemmas for some of the more important subprograms. The incorporation of these lemmas into the proof not only modularizes it but shortens it considerably as well. The reason we did not start out with the adalemma, approach is that we thought it would be too time-consuming and difficult within one year. But we were mistaken: our original proof strategy without adalemmas – while more straightforward – ultimately proved too time-consuming.

Apart from stress-testing SDVS, the verification project allowed us to enhance the SDVS Ada translator and add to the system three new proof commands that should facilitate long proofs.

Had we more time to work on the project, we would prove more adalemmas about some of the key subprograms (such as CMD_IN_HANDLER) and try to prove correctness assertions about more general schedulers and more realistic input conditions.

8 Appendix A – The Annotated Program

```
--SDVS comment: We deleted definitions, object declarations, task declarations,
--SDVS comment: and subprogram declarations that we did not need from several of
--SDVS comment: the packages below.  But we did not delete completely anything
--SDVS comment: from subprograms we use: if we did not need a particular statement
--SDVS comment: in a subprogram, we commented it out; the statement appears, but
--SDVS comment: only as a comment.  The exceptions to the rule are: (1) the
--SDVS comment: subprogram STORE_DATA_STRUC_IN_EEPROM in the package EEPROM_DATA
--SDVS comment: which bears no relation to the real one, (2) the function
--SDVS comment: CHECK_PARITY in the ASM_UTILITY package whose body is written in
--SDVS comment: 1750A assembly, (3) CHG_STATE and RETRIEVE_CUR_STATE in the
--SDVS comment: package MODE_STATE which are used to give us permission to write
--SDVS comment: to EEPROM, and (4) the procedure REPORT_SYSTEM_ERROR in the
--SDVS comment: package TM_DATA which is used to report errors. The packages
--SDVS comment: SERIAL_DIG_CMDS and APP_MSGS are presented in their entirety. For
--SDVS comment: our purposes, the packages GLOBAL_TYPES, CMDS_TYPES, and
--SDVS comment: TRKING_DATA_STRUC contain only type definitions and object
--SDVS comment: declarations used by SERIAL_DIG_CMDS and APP_MSGS.
--SDVS comment:
--SDVS comment: Our deletions by comment are prefixed by ``--SDVS delete:'' or by
--SDVS comment: ``--SDVS delete ('reason'),'' where 'reason' is one of the
--SDVS comment: reasons discussed in Section 4: for example, ``--SDVS delete
--SDVS comment: (1750A)'' means the deleted line includes a routine whose body is
--SDVS comment: written in 1750A assembly language. If the deleted line is
--SDVS comment: prefixed ``--SDVS delete:,'' the reason is given either within a
--SDVS comment: few lines preceding the deletion, or at the beginning of the
--SDVS comment: subprogram in which the statement appears.

with text_io; use text_io; --SDVS add for SDVS input and output
with integer_io; use integer_io; --SDVS add for SDVS input and output

procedure MSX_PROGRAM_FINAL_VERSION is

--SDVS comment: LONG_INTEGER and LONG_FLOAT are not standard Ada types. We provide
--SDVS comment: the SDVS_PACKAGE below for these type definitions.

package SDVS_PACKAGE is --SDVS add
  subtype LONG_INTEGER is INTEGER; --SDVS add
  subtype LONG_FLOAT is FLOAT; --SDVS add
end SDVS_PACKAGE; --SDVS add
use SDVS_PACKAGE; --SDVS add


-- $users:[na.adacode.bld1]global_types.ads
------------------------------------------------------------
--          GLOBAL_TYPES.ADS
------------------------------------------------------------
-- TYPE: Package Specification
--
-- PURPOSE : This package contains the types used throughout
--   the tracking processor software.
--
-- EXCEPTIONS:
```

```
--
-- NOTES:
--
-- CHANGE HISTORY:
--   mm/dd/yy    ?. ?????   Initial Version
--
------------------------------------------------------------
package GLOBAL_TYPES is

--SDVS comment: LONG_WORD type is used only in MEMORY_MANAGER and in ARRAY_OF_BLOCKS
--SDVS comment: It is not used in the code we will need.
--SDVS delete:  type LONG_WORD is range 0..2147483647;
--SDVS delete:  for LONG_WORD'size use 32;

  type WORD is range 0..65535;
  for WORD'size use 16;

  -- Bit Sliced Unsigned Integer Definitions.
  -- "WORD" is used for 16 bits and "BYTE" is used for 8 bits.

--SDVS comment:  types B15 .. B1 are never used in any part of the target code.

--SDVS delete:  type B15 is range 0..32767;
--SDVS delete:  for  B15'size use 15;

--SDVS delete:  type B14 is range 0..16383;
--SDVS delete:  for  B14'size use 14;

--SDVS delete:  type B13 is range 0..8191;
--SDVS delete:  for  B13'size use 13;

--SDVS delete:  type B12 is range 0..4095;
--SDVS delete:  for  B12'size use 12;

--SDVS delete:  type B11 is range 0..2047;
--SDVS delete:  for  B11'size use 11;

--SDVS delete:  type B10 is range 0..1023;
--SDVS delete:  for  B10'size use 10;

--SDVS delete:  type B9 is range 0..511;
--SDVS delete:  for  B9'size use 9;

  type BYTE is range 0..255;
  for BYTE'size use 8;

--SDVS delete:  type B7 is range 0..127;
--SDVS delete:  for  B7'size use 7;

--SDVS delete:  type B6 is range 0..63;
--SDVS delete:  for  B6'size use 6;

--SDVS delete:  type B5 is range 0..31;
--SDVS delete:  for  B5'size use 5;
```

```
--SDVS delete:  type B4 is range 0..15;
--SDVS delete:  for  B4'size use 4;

--SDVS delete:  type B3 is range 0..7;
--SDVS delete:  for  B3'size use 3;

--SDVS delete:  type B2 is range 0..3;
--SDVS delete:  for  B2'size use 2;

--SDVS delete:  type B1 is range 0..1;
--SDVS delete:  for B1'size use 1;

--sdvs comment: Only used in package TRKING_DATA_STRUC, in parts that we will not need.
--SDVS delete: type BYTE_INTEGER is range -128..127;
--SDVS delete: for BYTE_INTEGER'Size use 8;

--SDVS comment: Never used in any part of the target code.
--SDVS delete: type WORD_ARRAY_TYPE is array (INTEGER range <>) of WORD;

--SDVS comment: Never used in any part of the target code.
--SDVS delete: type BYTE_ARRAY_TYPE is array (INTEGER range <>) of BYTE;

--SDVS comment: Only used in the package TRKING_DATA_STRUC in parts that we will not need.
--SDVS delete: type ARRAY_OF_FLOAT_TYPE is array (INTEGER range <>) of FLOAT;


--SDVS comment: Only used in the package TRKING_DATA_STRUC, in parts that we will not need.
--SDVS delete: type ARRAY_OF_LONG_FLOAT_TYPE is array (INTEGER range <>) of LONG_FLOAT;


  type IO_STATUS_TYPE is array (0..15) of BOOLEAN;
--SDVS delete (pragma):   pragma pack(IO_STATUS_TYPE);

end GLOBAL_TYPES;


--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS
--SDVS The three package instantiations below are not part of the original
--SDVS MSX software and are not a part of the code translated by the SDVS
--SDVS translator. However these instantiations are needed to compile and
--SDVS execute the code with the Verdix compiler at Aerospace and are
--SDVS required by standard Ada compilers. They are needed for the ``get''
--SDVS for objects of type WORD in SERIAL_DIG_CMDS.CMD_IN_HANDLER,
--SDVS the ``put'' for objects of type INTEGER in
--SDVS EEPROM_DATA.STORE_DATA_STRUC_IN_EEPROM, and the ``put'' for
--SDVS objects of type BYTE in TM_DATA.MANAGE.REPORT_SYSTEM_ERROR.
--SDVS
--SDVS package  NEW_WORD_IO is new INTEGER_IO(GLOBAL_TYPES.WORD);  use NEW_WORD_IO;
--SDVS package  NEW_INTEGER_IO is new INTEGER_IO(INTEGER);  use NEW_INTEGER_IO;
--SDVS package  NEW_BYTE_IO is new INTEGER_IO(GLOBAL_TYPES.BYTE); use NEW_BYTE_IO;
--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS

-----------------------------------------------------------
--          ASM_UTILITY.ADS
-----------------------------------------------------------
```

```
-- TYPE: Package Specification
--
-- PURPOSE : This package contains the utility subroutines
--   written in the 1750A assembly language.
--
-- EXCEPTIONS:
--
-- NOTES:
--
-- CHANGE HISTORY:
--   10/16/91    B. Na      Initial Version
--
------------------------------------------------------------
--SDVS delete (with): with SYSTEM;

 package ASM_UTILITY is

--SDVS delete (1750A, pragma):   Pragma Foreign_Body ("Assembler");

--SDVS delete (1750A):  Function CHECK_PARITY (Address : in SYSTEM.ADDRESS)
--SDVS                                                          return BOOLEAN;
                Function CHECK_PARITY return BOOLEAN; --SDVS replace
--SDVS delete (1750A, pragma):   Pragma Linkage_Name (CHECK_PARITY ,
--SDVS                                                      "CHECK_PARITY");

end ASM_UTILITY;


--$users:[na.adacode.bld2]serial_dig_cmds.ads
------------------------------------------------------------
--          SERIAL_DIG_CMDS.ADS
------------------------------------------------------------
-- TYPE: Package Specification
--
-- PURPOSE : This package contains software to support the
--           interface with the serial digital command system.
--
--           The hierachy of subprograms in this package is
--           as follows:
--
--           I.    RET_CMD_COUNT (Called by the task APP_MSGS.BUILD)
--
--           II.   RET_CMD_BUF_AND_STATUS (Called by the task APP_MSGS.BUILD
--                 when the command count is > 0)
--
--           III.  CMD_IN_HANDLER (This is the standard interrupt handler for
--                 the short data command interrupt.  When a short data
--                 command interrupt occures, control is vectored to
--                 TP_STD_INT_PROC_SERVICE_SETUP, in turn
--                 TP_STD_INT_PROC_SERVICE_SETUP calls this routine to service
--                 the interrupt.)
--
-- EXCEPTIONS:
--
-- NOTES:
```

```
--
-- CHANGE HISTORY:
--   10/16/91    B. Na      Initial Version
--
----------------------------------------------------------
--SDVS delete (with): with GLOBAL_TYPES;

package SERIAL_DIG_CMDS is

  type CMD_BYTES_TYPE is record
    First_Byte  : GLOBAL_TYPES.BYTE;
    Second_Byte : GLOBAL_TYPES.BYTE;
    Third_Byte  : GLOBAL_TYPES.BYTE;
    Fourth_Byte : GLOBAL_TYPES.BYTE;
  end record;
--SDVS delete (pragma):  pragma pack(Cmd_Bytes_Type);

  Cmd_Buf_Size : constant INTEGER := 70;  -- 37.5 interrupts/sec

  type CMD_BUF_TYPE is array (1..Cmd_Buf_Size) of CMD_BYTES_TYPE;

  type CMD_STATUS_BUF_TYPE is array (1..Cmd_Buf_Size) of INTEGER;

  -- The following defines the possible command status values with the
  -- two exceptions:
  --      (1) Cmd_Lost is allowed only for the last (70th) command
  --          in the buffer.
  --      (2) Cmd_Lost and Cmd_FIFO_Not_Empty are ORed to the rest of
  --          other status values.

  Cmd_OK              : constant INTEGER := 0;
  App_Msg_Dropped     : constant INTEGER := 1;
  Bad_Checksum        : constant INTEGER := 2;
  Flushed_prior_Build : constant INTEGER := 3;
  Invalid_Data_Struct_Msg_ID : constant INTEGER := 4;
  Invalid_Starting_Cmd       : constant INTEGER := 5;
  App_Msg_Overwritten : constant INTEGER := 6;
                                             -- These are checked by
                                             -- by SERIAL_DIG_CMDS.
  Cmd_Bad_Parity      : constant INTEGER := 7;  -- Parity of the command is
                                             -- not odd.
  Cmd_Lost            : constant INTEGER := 8;  -- New command is lost
                                             -- because the command buffer
                                             -- is full.
  Cmd_FIFO_Not_Empty  : constant INTEGER := 16; -- The command in FIFO is not
                                             -- empty after one command
                                             -- (4 bytes) is read.

  -- function/procedure specifications

  function RET_CMD_COUNT return INTEGER;

  procedure RET_CMD_BUF_AND_STATUS
          (Cmd_Buf              : out CMD_BUF_TYPE;
           Cmd_Status_Buf       : out CMD_STATUS_BUF_TYPE;
```

```
            Cmd_Count            : out INTEGER;
            FIFO_Not_Empty_Count : out INTEGER;
            IO_Status_Word       : out GLOBAL_TYPES.WORD;
            Lost_Cmd             : out BOOLEAN);


--SDVS comment: The function RET_MSG_ID has been added
--SDVS comment: for the scheduler. It returns the message identifier for
--SDVS comment: the next block of commands in the input encoding a message.
  function RET_MSG_ID return GLOBAL_TYPES.BYTE; --SDVS add

  procedure CMD_IN_HANDLER;
--SDVS delete (pragma):  pragma linkage_name(CMD_IN_HANDLER, "CMD_IN_HANDLER");

end SERIAL_DIG_CMDS;

-- $users:[na.adacode.bld2.longload.ds]cmds_types.ads
------------------------------------------------------------
--          CMDS_TYPES.ADS
------------------------------------------------------------
-- TYPE: Package Specification
--
-- PURPOSE : This package defines the data types that are
-- commonly used by the Process Commands group packages.
--
-- EXCEPTIONS:
--
-- NOTES:
--
-- CHANGE HISTORY:
--   mm/dd/yy    B. Na      Initial version
--
------------------------------------------------------------
package CMDS_TYPES is

  App_Msg_Max            : constant INTEGER := 80;
--SDVS comment: The predefined type POSITIVE is not implemented in SDVS.
--SDVS delete:  subtype INDEX_SUBTYPE  is POSITIVE range 1..App_Msg_Max;
  subtype INDEX_SUBTYPE  is INTEGER range 1..App_Msg_Max;--SDVS replace
--SDVS comment: SDVS does not currently handle unconstrained array types.
--SDVS delete (unconstrained array):  type APP_MSG_OUT_TYPE  is
--SDVS cont:         array (INDEX_SUBTYPE range<>) of INTEGER;
  type APP_MSG_OUT_TYPE  is array (1 .. App_Msg_Max) of INTEGER; --SDVS replace
  Load_Msg_Max           : constant INTEGER := 127;
  type LOAD_MSG_TYPE     is array (1..Load_Msg_Max) of INTEGER;

end CMDS_TYPES;


------------------------------------------------------------
--          MODE_STATE.ADS
------------------------------------------------------------
-- TYPE: Package Specification
--
-- PURPOSE : BOGUS MODE_STATE!
```

```
--
-- EXCEPTIONS:
--
-- NOTES:
--
-- CHANGE HISTORY:
--   mm/dd/yy    ?. ?????   Initial Version
--
-----------------------------------------------------------
package MODE_STATE is

  type MODE_STATE_TYPE is (Power_Up, Init_Setup, Init_EEPROM_Write,
                           Init_Memory_Load, Init_Memory_Dump,
                           Tracking_Setup, Tracking);

  type STATUS_OF_REQUEST_TYPE is (State_Granted, State_Denied);

  procedure PERFORM_SOFT_RESET;

  function RETRIEVE_CUR_STATE return MODE_STATE_TYPE;

  procedure CHG_STATE(Mode_State_Req    : in  MODE_STATE_TYPE;
                      Status_Of_Request : out STATUS_OF_REQUEST_TYPE);

end MODE_STATE;

-- $users:[na.adacode.bld2.longload.ds]eeprom_data.ads
-----------------------------------------------------------
--          EEPROM_DATA.ADS
-----------------------------------------------------------
-- TYPE: Package Specification
--
-- PURPOSE:  This package stores and retrieves EEPROM data.
--           There are three types of EEPROM data stored and
--           retrieved by this package:
--
--                1.  data structure
--                2.  specific memory (EEPROM) load data
--                3.  miscellaneous system variable data
--
--           The hierarchy of external subprograms in this package
--           is as follows:
--
--           I.    STORE_DATA_STRUC_IN_EEPROM (called by APP_MSGS.PROCESS_MSG
--                                and ARRAY_OF_BLOCKS.PROCESS_MSGS_IN_BLOCKS)
--
--           II.   RET_DATA_STRUC_IN_EEPROM (called by TRKING_PARAMS.INIT_PARMS)
--
--           III.  RET_SYS_CONFIG (called by APP_MSGS.PROCESS_MSG on powerup)
--
--           IV.   WRITE_LOAD_DATA_TO_EEPROM (called by APP_MSGS.PROCESS_MSG
--                                 and ARRAY_OF_BLOCKS.PROCESS_MSGS_IN_BLOCKS)
--
-- EXCEPTIONS:
--
```

```
-- NOTES:
--
-- CHANGE HISTORY:  01/31/92    B. Na      Initial Version
--
------------------------------------------------------------
--SDVS delete (with): with GLOBAL_TYPES;
--SDVS delete (with): with CMDS_TYPES;


package EEPROM_DATA is

  procedure STORE_DATA_STRUC_IN_EEPROM
       (Array_Of_Words           : in     CMDS_TYPES.APP_MSG_OUT_TYPE;
        Data_Struc_ID            : in     INTEGER;
        Data_Struc_Length        : in     INTEGER);

end EEPROM_DATA;



--
--SDVS delete (with): with GLOBAL_TYPES;

package TRKING_DATA_STRUC is

  -- This package defines types to support data structures specified in the
  -- TP/CP IDS.  The following table is a list of the data structure message
  -- IDs and associated name.
  ---------------------------------------------------------------------------
  --   1    Beacon alignment first  object
  --   2    Beacon alignment second object
  --   3    UVISI  alignment visible NFOV imager
  --   4    UVISI  alignment visible WFOV imager
  --   5    UVISI  alignment UV      NFOV imager
  --   6    UVISI  alignment UV      WFOV imager
  --   7    OSDP   alignment

  --   8    Initial Covariance
  --   9    Covariance Q-ing

  --  10    AP propagation time
  --  11    Filter lag time

  --  12    Beacon receiver measurement noise for frequency # 1
  --  13    Beacon receiver measurement noise for frequency # 2
  --  14    UVISI           measurement noise visible NFOV imager
  --  15    UVISI           measurement noise visible WFOV imager
  --  16    UVISI           measurement noise UV      NFOV imager
  --  17    UVISI           measurement noise UV      WFOV imager
  --  18    OSDP            measurement noise

  --  19    UVISI data association processing parameters
  --  20    Roll angle bias
  --  21    Sensor data priority
  --  22    Filter loss of track rules
--  23   Sensor data acceptance threshold
--  24   UVISI 4 Hz Status
--  25   UVISI 4 Hz angle threshold
--  26   Priority level loss of track switch times
--  27   Raw data alpha - beta filter
--  28   Year
--  29   Roll Law Reference Vector

--  50   Event time/selection
--  51   Y-Z offset
--  52   Event stage times
--  53   Stage 1 polynomial coefficients
--  54   Stage 2 polynomial coefficients
--  55   Stage 1 sinusoidal curve fit coefficients
--  56   Stage 2 sinusoidal curve fit coefficients
--  57   Satellite parameters
--  58   Lat., Long., Alt., coefficients
--  59   Az., & tangent height coefficients
--  60   ECI vector
--  61   Az & EL coefficients
--  62   Beacon frequencies
--  63   Attitude Quaternions and rates
--  64   Attitude Quaternions - zero rates
--  65   Object Deployment Parameters
--  66   Time reference scan parameters
--  67   Geometric scan paramters

--  80   Change event stop time
--  81   Change sensor selection
--  82   Change pointing selection
--  83   New roll law selection number
--  84   Filter UVISI enable/disable
--  85   Change UVISI data association algorithm to use
--  86   Change stage 1 time
--  87   Change stage 2 time
--  88   Change current event Y-Z offset
--  89   New beacon frequency
--  90   Beacon object number to switch to
--  91   Change attitude quaternion and rates for current event
--  92   Change attitude quaternion only - zero rates for current event
--  93   Turn time filtering on/off
-------------------------------------------------------------------------

Num_Data_Struc : constant INTEGER := 109;

-- This table lists the number of words in the message for
-- a data structure load message.  The data structure ID is used as an
-- index into the table.  ID 1 thru 49 are reserved for current/futrue
-- event, 50 thru 79 are reserved for next event and 80 thru 109 are
-- are reserved for current data structures.

-- This table is used by APP_MSGS, ARRAY_OF_BLOCKS, and TRKING_PARAMS.

type WORDS_FOR_DATA_STRUC_TYPE is array (1..Num_Data_Struc) of integer;
Words_For_Data_Struc : constant WORDS_FOR_DATA_STRUC_TYPE :=
      (20, 20, 20, 20, 20, 20, 20, 14, 14, 04,
       04, 06, 06, 04, 04, 04, 04, 04, 11, 04,
       04, 03, 06, 03, 05, 05, 06, 03, 08, 00,
       00, 00, 00, 00, 00, 00, 00, 00, 00, 00,
       00, 00, 00, 00, 00, 00, 00, 00, 00,

       15, 06, 08, 57, 57, 78, 78, 19, 08, 06,
       09, 06, 07, 16, 10, 11, 11, 22, 00, 00,
       00, 00, 00, 00, 00, 00, 00, 00, 00, 00,

       05, 03, 03, 03, 03, 03, 05, 05, 06, 05,
       03, 16, 10, 03, 00, 00, 00, 00, 00, 00,
       00, 00, 00, 00, 00, 00, 00, 00, 00, 00);

end TRKING_DATA_STRUC;



------------------------------------------------------------
--          TM_DATA.ADS
------------------------------------------------------------
-- TYPE: Package Specification
--
-- PURPOSE:  This buffer package stores new telemetry data,
--           such as system errors, pointing or tracking data
--           for the last half second, UVISI gate results and
--           the last transmitted AP and Broadcast messages.
--
--           "Document something about command-out counter."
--
--           The buffered data is then retrieved by TM.COLLECT,
--           once a second, for reporting in housekeeping
--           telemetry.
--
-- EXCEPTIONS:
--
-- NOTES:    This package is expected to be in the same Address
--           State as TM.  Sys-Error-Buffer is a package level
--           variable and passed by reference.
--
--
------------------------------------------------------------

package TM_DATA is



-- MANAGE TASK TYPE

--SDVS delete (task):  task type MANAGE_TYPE is
--SDVS delete (task):    entry REPORT_SYSTEM_ERROR(Sys_Error_Code :
--SDVS                                                in GLOBAL_TYPES.BYTE);
```

```
--SDVS delete (task):  end MANAGE_TYPE;
--SDVS delete (task):  for MANAGE_TYPE'storage_size use 300 ;
--SDVS delete (task):  MANAGE : MANAGE_TYPE;


    package MANAGE is  --SDVS replacement

      procedure REPORT_SYSTEM_ERROR(Sys_Error_Code :
                                  in GLOBAL_TYPES.BYTE); --SDVS replacement

    end MANAGE; --SDVS replacement

  end TM_DATA;
```

```
-- $users:[na.adacode.bld2.longload.ds]app_msgs.ads
------------------------------------------------------------
--          APP_MSGS.ADS
------------------------------------------------------------
-- TYPE: Package Specification
--
-- PURPOSE : This package contains software to build and
--           process the application messages for the serial digital
--           commands buffered by SERIAL_DIG_CMDS.
--
--   This package consists of following tasks and external
--   procedures and function:
--
--   I.    BUILD  (This task runs at a 2 Hz rate.)
--
--   II.   PROCESS_MSG  (This task runs without any delay but pends at the
--                       guarded entry MANAGE.MSG_RETRIEVAL.RET_NEXT_MSG.)
--
--   III.  MANAGE_MSG_RETRIEVAL (This task rendezvous with PROCESS_MSG when
--                               the application message count is greater
--                               than zero.)
--
--   IV.   Other Procedures or Functions
--           A.  NEW_TM_DATA
--           B.  RET_LATEST_CMDS
--           C.  RET_INIT_CMD_LAST_APP_MSG
--
-- EXCEPTIONS:
--
-- NOTES:
--
-- CHANGE HISTORY:
--   10/16/91    B. Na      Initial Version
--   mm/dd/yy    B. Na      Build 2 Changes
--
------------------------------------------------------------
--SDVS delete (with): with SERIAL_DIG_CMDS;
--SDVS delete (with): with GLOBAL_TYPES;

package APP_MSGS is

  Max_Num_Latest_Cmds : constant INTEGER := 40;
  type LATEST_CMD_BUF_TYPE is array (1..Max_Num_Latest_Cmds) of
                                        Serial_Dig_Cmds.Cmd_Bytes_Type;
  type LATEST_STATUS_BUF_TYPE is array (1..Max_Num_Latest_Cmds) of
                                        INTEGER;

  type LATEST_CMD_TM_TYPE is record
    Cmd_Buf              : LATEST_CMD_BUF_TYPE;
    Status_Buf           : LATEST_STATUS_BUF_TYPE;
    Fresh_Cmd_Count      : INTEGER;
    Cmd_Total            : INTEGER;
    IO_Status_Word       : GLOBAL_TYPES.WORD;
    Rejected_Cmd_Count   : INTEGER;
    FIFO_Not_Empty_Count : INTEGER;
    Saved_Sat_Subsys_Config : GLOBAL_TYPES.WORD;
  end record;

  type LAST_APP_MSG_TM_TYPE is record
    Starting_Cmd         : SERIAL_DIG_CMDS.CMD_BYTES_TYPE;
    Cmd_Total            : INTEGER;
    IO_Status_Word       : GLOBAL_TYPES.WORD;
    Rejected_Cmd_Count   : INTEGER;
    FIFO_Not_Empty_Count : INTEGER;
    Saved_Sat_Subsys_Config : GLOBAL_TYPES.WORD;

  end record;


  -- function/procedure specifications
--SDVS comment: The function and two procedures that follow are never
--SDVS comment: called in this target code. They are
--SDVS comment: called by TM.COLLECT for telemetry reports on the system.
--SDVS delete:  function NEW_TM_DATA return BOOLEAN;

--SDVS delete:   procedure RET_LATEST_CMDS (Latest_Cmd_TM   :
--SDVS delete:                                          out LATEST_CMD_TM_TYPE);

--SDVS delete:   procedure RET_INIT_CMD_LAST_APP_MSG
--SDVS delete:                  (Last_App_Msg_TM : out LAST_APP_MSG_TM_TYPE);


--SDVS comment: The three procedures below were originally tasks and did
--SDVS comment: not appear in the visible part of the package APP_MSGS.
--SDVS comment: We have to include them in the visible part
--SDVS comment: of this package because they will be called by the main program.
  procedure BUILD; --SDVS replace: Replaces task declaration type in body of APP_MSGS.
  procedure MANAGE_MSG_RETRIEVAL(App_Msg_Out : out CMDS_TYPES.APP_MSG_OUT_TYPE);
```

```
-- SDVS replace: Replaces task declaration type in body of APP_MSGS.
  procedure PROCESS_MSG; --SDVS replace: Replaces task declaration type
--SDVS in body of APP_MSGS.

--SDVS comment: The function RET_MSG_LENGTH returns the length of the next block
--SDVS comment: of commands in the input based on the message identifier. We
--SDVS comment: added it for the scheduler.
function RET_MSG_LENGTH(Msg_Id: GLOBAL_TYPES.BYTE) return INTEGER; --SDVS add

end APP_MSGS;



--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS
--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS
--SDVS comment:              BEGINNING OF BODY DECLARATIONS
--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS
--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS



--SDVS delete (1750A):             MODULE ASM_UTILITY

--SDVS delete (1750A): * Routines in this module !

--SDVS delete (1750A):  DEFINE CHECK_PARITY          * perform odd-parity check on
--SDVS delete (1750A):                              * Cmd_Bytes_Type.
--SDVS delete (1750A):
--SDVS delete (1750A): USRCODE      CSECT,I
--SDVS delete (1750A): ***********************************************************
--SDVS delete (1750A): *
--SDVS delete (1750A): *    Function Check_Parity (Address : in SYSTEM.ADDRESS)
--SDVS delete (1750A): *                                      return BOOLEAN
--SDVS delete (1750A): *    1.  Read 4 words stored starting at Address.
--SDVS delete (1750A): *    2.  Count the number of 1's in the lower bytes
--SDVS delete (1750A): *        (bit 8 thru 15) of words.
--SDVS delete (1750A): *    3.  If odd number of 1's, then return 1 (True).
--SDVS delete (1750A): *        Otherwise, return 0 (False).
--SDVS delete (1750A): *
--SDVS delete (1750A): ****************************************************************
--SDVS delete (1750A): CHECK_PARITY  EQU     $
--SDVS delete (1750A): *
--SDVS delete (1750A): * Get the Parameters off the stack
--SDVS delete (1750A): *   SP => "Address"
--SDVS delete (1750A): *
--SDVS delete (1750A):     L       R4,0,R15              * get the parameter (Address) of
--SDVS delete (1750A):                                   *   Cmd_Bytes_Type off the stack
--SDVS delete (1750A):     PSHM    R4,R6                 * save R4 thru R6
--SDVS delete (1750A):     SR      R2,R2                 * R2 <- init num_of_1's in
--SDVS delete (1750A):                                   * Cmd_Bytes_Type data
--SDVS delete (1750A):     LIM     R6,4                  * Cmd_Bytes_Type data occupies
--SDVS delete (1750A):                                   * 4 words
--SDVS delete (1750A): BITWORD      EQU     $
--SDVS delete (1750A):     LIM     R3,8                  * interested in only
--SDVS delete (1750A):                                     lower 8 bits
```

```
--SDVS delete (1750A):     L       R5,0,R4             * get the cmd byte stored
--SDVS delete (1750A):                                 * in bit 8 thru 15
--SDVS delete (1750A): BITCHEK       EQU    $
--SDVS delete (1750A):     TBR     8,R5                * if bit 8 is 0
--SDVS delete (1750A):     JC      2,BITZERO           *   skip incrementing num_of_1's
--SDVS delete (1750A):                                 * else
--SDVS delete (1750A):     AIM     R2,1                *   increment num_of_1's
--SDVS delete (1750A): BITZERO       EQU    $
--SDVS delete (1750A):     SLL     R5,1                * shift logical left once
--SDVS delete (1750A):     SOJ     R3,BITCHEK          * repeat for 8 bits
--SDVS delete (1750A):     AIM     R4,1                * get the next word
--SDVS delete (1750A):     SOJ     R6,BITWORD          * repeat for 4 words
--SDVS delete (1750A):
--SDVS delete (1750A):     ANDM    R2,HEX(0001)        * we only care whether odd
--SDVS delete (1750A):                                 * or even number of 1's
--SDVS delete (1750A):     POPM    R4,R6               * restore R4 thru R6
--SDVS delete (1750A):     LR      R3,R0
--SDVS delete (1750A): * note: JC uses an index register which is R1...R15 (i.e. not R0)
--SDVS delete (1750A):               JC      7,0,R3              * GOTO RETURN ADDRESS
--SDVS delete (1750A): *
--SDVS delete (1750A): * THATS ALL FOLKS
--SDVS delete (1750A): *
--SDVS delete (1750A): * ITS assembler doesn't predefine R0..R15
--SDVS delete (1750A): *
--SDVS delete (1750A): R0            EQU    0
--SDVS delete (1750A): R1            EQU    1
--SDVS delete (1750A): R2            EQU    2
--SDVS delete (1750A): R3            EQU    3
--SDVS delete (1750A): R4            EQU    4
--SDVS delete (1750A): R5            EQU    5
--SDVS delete (1750A): R6            EQU    6
--SDVS delete (1750A): R7            EQU    7
--SDVS delete (1750A): R8            EQU    8
--SDVS delete (1750A): R9            EQU    9
--SDVS delete (1750A): R10           EQU    10
--SDVS delete (1750A): R11           EQU    11
--SDVS delete (1750A): R12           EQU    12
--SDVS delete (1750A): R13           EQU    13
--SDVS delete (1750A): R14           EQU    14
--SDVS delete (1750A): R15           EQU    15
--SDVS delete (1750A): ALL           EQU    7
--SDVS delete (1750A):               END


package body ASM_UTILITY is --SDVS add

 function CHECK_PARITY return BOOLEAN is --SDVS add
  begin --SDVS add
   return true; --SDVS add
  end CHECK_PARITY; --SDVS add

end ASM_UTILITY; --SDVS add

------------------------------------------------------------
--         INTRINSIC_FUNCTIONS.ADS
```

```
------------------------------------------------------------
-- TYPE: Package Specification
--
-- PURPOSE : this package instantiates the Tartan intrinsic functions to
-- perform XIO functions and bit operations.
--
-- EXCEPTIONS:
--
-- NOTES:
--
-- CHANGE HISTORY
-- 02/14/92   R. Pham     Initial Version
-- 02/28/92   R. Pham     Instantiate Shift functions using UNSIGNED_WORD as
--                        result type
-- 03/02/92   R. Pham     Instantiate functions using UNSIGNED_BYTE
-- 03/05/92   S. Hutton   Modified for tracking processor
------------------------------------------------------------

--SDVS comment: This package instantiates generic functions from the Tartan
--SDVS comment: supplied INTRISICS package. The functions perform bit
--SDVS comment: manipulations. We needed only a few of the instantiated
--SDVS comment: functions. To compile and execute the code with the
--SDVS comment: Verdix compiler at Aerospace, we supplied Ada code for
--SDVS comment: the functions we needed. In the final SDVS proofs, this
--SDVS comment: code was not used: we used adalemmas to characterize
--SDVS comment: their behavior. These adalemmas were not proved.


--SDVS delete (with): with GLOBAL_TYPES;
--SDVS delete (with): with INTRINSICS;   use INTRINSICS;

package INTRINSIC_FUNCTIONS is

--SDVS delete:  function  AND_I              is new ANDi   (INTEGER,
--SDVS delete:                                              INTEGER,
--SDVS delete:                                              INTEGER);
  function  AND_I(X, Y: INTEGER) return INTEGER; --SDVS replace above declaration

--SDVS comment: Needed for masking in PROCESS_MSG_TYPE.
--SDVS delete: function  AND_WORD          is new ANDi   (GLOBAL_TYPES.WORD,
--SDVS delete:                                              GLOBAL_TYPES.WORD,
--SDVS delete:                                              GLOBAL_TYPES.WORD);
  function  AND_WORD(X, Y: GLOBAL_TYPES.WORD) return GLOBAL_TYPES.WORD; --SDVS replace


  -- The comment in the spec says 16 or 32-bit logical and.  The comment is
  -- not next to the operand or result type.  Thus, bytes can be used and the
  -- system will generate correct code.
--SDVS comment: Needed for masking in Build_Type (only use in the code).
--SDVS delete:  function  AND_BYTE          is new ANDi   (GLOBAL_TYPES.BYTE,
--SDVS delete:                                              GLOBAL_TYPES.BYTE,
--SDVS delete:                                              GLOBAL_TYPES.BYTE);
  function  AND_BYTE(X, Y: GLOBAL_TYPES.BYTE)
                                      return GLOBAL_TYPES.BYTE; --SDVS replace
  -- "or" is a reserved word.
--SDVS delete: function  OR_I               is new ORi    (INTEGER,
--SDVS delete:                                             INTEGER,
--SDVS delete:                                             INTEGER);
  function  OR_I(X, Y: INTEGER) return INTEGER;--SDVS replace


--SDVS delete:  function  OR_WORD           is new ORi    (GLOBAL_TYPES.WORD,
--SDVS delete:                                             GLOBAL_TYPES.WORD,
--SDVS delete:                                             GLOBAL_TYPES.WORD);
  function  OR_WORD(X, Y: GLOBAL_TYPES.WORD)
                                      return GLOBAL_TYPES.WORD; --SDVS replace




--SDVS delete:  function  XOR_WORD          is new XORi   (GLOBAL_TYPES.WORD,
--SDVS delete:                                             GLOBAL_TYPES.WORD,
--SDVS delete:                                             GLOBAL_TYPES.WORD);
  function  XOR_WORD(X, Y: GLOBAL_TYPES.WORD)
                                      return GLOBAL_TYPES.WORD; --SDVS replace


  -- 16-bit shift logical.  For shift logical count in register, if |N|>16,
  -- the fixed point overflow occurs.
--SDVS delete:  function  SHIFT_LOGICAL_WORD is new SL     (GLOBAL_TYPES.WORD,
--SDVS delete:                                             GLOBAL_TYPES.WORD);
  function  SHIFT_LOGICAL_WORD(X: GLOBAL_TYPES.WORD; Y: INTEGER)
                                      return GLOBAL_TYPES.WORD; --SDVS replace

end INTRINSIC_FUNCTIONS;




--SDVS comment: We wrote code for the functions in the INTRINSIC_FUNCTIONS
--SDVS comment: package below to test the target software on the Verdix
--SDVS comment: compiler. Our proofs do not use the code. We characterized
--SDVS comment: these functions by adalemmas for the code.
package body INTRINSIC_FUNCTIONS is         --SDVS add
use GLOBAL_TYPES; --SDVS add
        function AND_I (X, Y: INTEGER) return INTEGER is --SDVS add
                TEMPX, TEMPY, RET: INTEGER; --SDVS add
                POWER: INTEGER; --SDVS add
                begin --SDVS add
                        TEMPX := X; --SDVS add
                        TEMPY := Y; --SDVS add
                        RET := 0; --SDVS add
                        POWER := 0; --SDVS add
                        while (TEMPX /= 0) or (TEMPY /= 0) loop --SDVS add
                          RET :=
                           RET + ((TEMPY mod 2)*(TEMPX mod 2))*(2 ** POWER); --SDVS add
                          TEMPX := TEMPX / 2; TEMPY := TEMPY / 2; --SDVS add
                          POWER := POWER + 1; --SDVS add
                end loop; --SDVS add
                return RET; --SDVS add
             end AND_I; --SDVS add

function AND_WORD (X, Y: GLOBAL_TYPES.WORD)
                          return GLOBAL_TYPES.WORD is --SDVS add
        TEMPX, TEMPY, RET: GLOBAL_TYPES.WORD; --SDVS add
        POWER: GLOBAL_TYPES.WORD;  --SDVS add
        begin --SDVS add
                TEMPX := X; --SDVS add
                TEMPY := Y; --SDVS add
                RET := 0; --SDVS add
                POWER := 0; --SDVS add
                while (TEMPX /= 0) or (TEMPY /= 0) loop --SDVS add
                  RET :=
                    RET + ((TEMPY mod 2)*(TEMPX mod 2))*(2 ** POWER); --SDVS add
                  TEMPX := TEMPX / 2; TEMPY := TEMPY / 2; --SDVS add
                  POWER := POWER + 1; --SDVS add
                end loop; --SDVS add
                return RET; --SDVS add
             end AND_WORD; --SDVS add

function AND_BYTE (X, Y: GLOBAL_TYPES.BYTE)
                          return GLOBAL_TYPES.BYTE is --SDVS add
        TEMPX, TEMPY, RET: GLOBAL_TYPES.BYTE; --SDVS add
        POWER: GLOBAL_TYPES.BYTE;  --SDVS add
        begin --SDVS add
                TEMPX := X; --SDVS add
                TEMPY := Y; --SDVS add
                RET := 0; --SDVS add
                POWER := 0; --SDVS add
                while (TEMPX /= 0) or (TEMPY /= 0) loop --SDVS add
                  RET :=
                    RET + ((TEMPY mod 2)*(TEMPX mod 2))*(2 ** POWER); --SDVS add
                  TEMPX := TEMPX / 2; TEMPY := TEMPY / 2; --SDVS add
                  POWER := POWER + 1; --SDVS add
                end loop; --SDVS add
                return RET; --SDVS add
             end AND_BYTE; --SDVS add

function OR_I (X, Y:  INTEGER) return INTEGER is --SDVS add
        TEMPX, TEMPY, RET: INTEGER; --SDVS add
        POWER:  INTEGER; --SDVS add
        begin --SDVS add
                TEMPX := X; --SDVS add
                TEMPY := Y; --SDVS add
                RET := 0; --SDVS add
                POWER := 0; --SDVS add
                while (TEMPX /= 0) or (TEMPY /= 0) loop --SDVS add
                  RET := RET + (TEMPX mod 2 + TEMPY mod 2 -  --SDVS add
                    ((TEMPY mod 2)*(TEMPX mod 2)))*(2 ** POWER); --SDVS add
                  TEMPX := TEMPX / 2; TEMPY := TEMPY / 2; --SDVS add
                  POWER := POWER + 1; --SDVS add
                end loop; --SDVS add
                RETURN RET; --SDVS add
             end OR_I; --SDVS add

     function OR_WORD (X, Y:  GLOBAL_TYPES.WORD)
                                  return GLOBAL_TYPES.WORD is --SDVS add
            TEMPX, TEMPY, RET: GLOBAL_TYPES.WORD; --SDVS add
            POWER: GLOBAL_TYPES.WORD;  --SDVS add
            begin --SDVS add
                    TEMPX := X; --SDVS add
                    TEMPY := Y; --SDVS add
                    RET := 0; --SDVS add
                    POWER := 0; --SDVS add
                    while (TEMPX /= 0) or (TEMPY /= 0) loop --SDVS add
                      RET := RET + (TEMPX mod 2 + TEMPY mod 2 -  --SDVS add
                            ((TEMPY mod 2)*(TEMPX mod 2)))*(2 ** POWER); --SDVS add
                      TEMPX := TEMPX / 2; TEMPY := TEMPY / 2; --SDVS add
                      POWER := POWER + 1; --SDVS add
                    end loop; --SDVS add
                    RETURN RET; --SDVS add
                 end OR_WORD; --SDVS add

     function SHIFT_LOGICAL_WORD (X: GLOBAL_TYPES.WORD; Y: INTEGER)
                                  return GLOBAL_TYPES.WORD is --SDVS add
            TEMPX:GLOBAL_TYPES.WORD; --SDVS add
            begin --SDVS add
            TEMPX := X; --SDVS add
            if Y >= 0 then --SDVS add
                    for I in 1..Y loop --SDVS add
                      if TEMPX >= 32768 then TEMPX := TEMPX - 32768; --SDVS add
                      end if; --SDVS add
                      TEMPX := TEMPX*2; --SDVS add
                    end loop; --SDVS add
                    return TEMPX; --SDVS add
                    else --SDVS add
                      for I in 1..(-Y) loop --SDVS add
                        TEMPX := TEMPX / 2; --SDVS add
                    end loop; --SDVS add
                    return TEMPX; --SDVS add
            end if; --SDVS add
            end SHIFT_LOGICAL_WORD;                 --SDVS add

     function XOR_WORD (X, Y:  GLOBAL_TYPES.WORD)
                                    return GLOBAL_TYPES.WORD is --SDVS add
            TEMPX, TEMPY, RET: GLOBAL_TYPES.WORD; --SDVS add
            POWER: GLOBAL_TYPES.WORD; --SDVS add
            begin --SDVS add
                    TEMPX := X; --SDVS add
                    TEMPY := Y; --SDVS add
                    RET := 0; --SDVS add
                    POWER := 0; --SDVS add
                    while (TEMPX /= 0) or (TEMPY /= 0) loop --SDVS add
                      RET := RET + (TEMPX mod 2 + TEMPY mod 2 -  --SDVS add
                         ((TEMPY mod 2)*(TEMPX mod 2)))*(2 ** POWER); --SDVS add
                      TEMPX := TEMPX / 2; TEMPY := TEMPY / 2; --SDVS add
                      POWER := POWER + 1; --SDVS add
                    end loop; --SDVS add
```

```
                        RETURN RET; --SDVS add
                      end XOR_WORD; --SDVS add

end INTRINSIC_FUNCTIONS; --SDVS add


--$users:[na.adacode.bld2.longload.ds.chg]serial_dig_cmds.adb
------------------------------------------------------------
--          SERIAL_DIG_CMDS.ADB
------------------------------------------------------------
-- TYPE: Package Body
--
-- PURPOSE : This package contains software to support the
--           interface with the serial digital command system.
--
--           The hierachy of subprograms in this package is
--           as follows:
--
--           I.    RET_CMD_COUNT (Called by the task APP_MSGS.BUILD)
--
--           II.   RET_CMD_BUF_AND_STATUS (Called by the task APP_MSGS.BUILD
--                 when the command count is > 0)
--
--           III.  CMD_IN_HANDLER (This is the standard interrupt handler for
--                 the short data command interrupt.  When a short data
--                 command interrupt occures, control is vectored to
--                 TP_STD_INT_PROC_SERVICE_SETUP, in turn
--                 TP_STD_INT_PROC_SERVICE_SETUP calls this routine to service
--                 the interrupt.)
--                 A.ASM_UTILITY.CHECK_PARITY
--
-- EXCEPTIONS:
--
-- NOTES:
--
-- CHANGE HISTORY:
--   10/16/91    B. Na      Initial Version
--   yy/mm/dd    B. Na      Build 2 Changes
--
------------------------------------------------------------
--SDVS delete (with): with GLOBAL_TYPES;
--SDVS delete (with): with ARTCLIENT;
--SDVS delete (with): with UNCHECKED_CONVERSION;
--SDVS delete (with): with INTRINSIC_FUNCTIONS;
use INTRINSIC_FUNCTIONS;
--SDVS delete (with): with MEMORY_MANAGER;
--SDVS delete (with): with ASM_UTILITY;


package body SERIAL_DIG_CMDS is

  -- Holds the commands read from the FIFO and not yet retrieved by
  -- APP_MSGS.BUILD.  Filled with the commands as they received by
  -- CMD_IN_HANDLER.  Cmd_Count is used as an index into the buffer.
  -- If Cmd_Count = N, the following depicts the command buffer.
--      +--------------------+--------------------+
--      |   First Byte       |   Second Byte      |
--  (1) |--------------------+--------------------| Command 1
--      |   Third Byte       |   Fourth Byte      |
--      +--------------------+--------------------+
--      |   First Byte       |   Second Byte      |
--  (2) |--------------------+--------------------| Command 2
--      |   Third Byte       |   Fourth Byte      |
--      +--------------------+--------------------+
--      |   First Byte       |   Second Byte      |
--  (3) |--------------------+--------------------| Command 3
--      |   Third Byte       |   Fourth Byte      |
--      +--------------------+--------------------+
--                           .
--                           .
--                           .
--      |   First Byte       |   Second Byte      |
--  (N) |--------------------+--------------------| Command N
--      |   Third Byte       |   Fourth Byte      |
--      +--------------------+--------------------+
--                           .
--                           .
--                           .
--      |                    |                    |
-- (70) |--------------------+--------------------|
--      |                    |                    |
--      +--------------------+--------------------+
Cmd_Buf          : CMD_BUF_TYPE;

-- Holds the command statuses that are not yet retrieved by
-- APP_MSGS.BUILD.  Cmd_Count is used as an index into the buffer.
-- If Cmd_Count = N, the following depicts the status buffer.
--      +-----------------------------------------+
--  (1) |             Command 1 Status            |
--      +-----------------------------------------+
--  (2) |             Command 2 Status            |
--      +-----------------------------------------+
--  (3) |             Command 3 Status            |
--      +-----------------------------------------+
--                           .
--                           .
--                           .
--  (N) |             Command N Status            |
--      +-----------------------------------------+
--                           .
--                           .
--                           .
--      |                                         |
--      +-----------------------------------------+
--  (70)|                                         |
--      +-----------------------------------------+
Cmd_Status_Buf : CMD_STATUS_BUF_TYPE;

-- The following command status values are possible while the
-- commands are being received and buffered by SERIAL_DIG_CMDS:
  --      Cmd_OK = 0,
  --      Cmd_Bad_Parity = 7,
  --      Cmd_Lost = 8,
  --      Cmd_FIFO_Not_Empty = 16
  --      Cmd_FIFO_Not_Empty and Cmd_Bad_Parity = 16 + 7
  --      Cmd_Lost and Cmd_Bad_Parity = 8 + 7
  -- Cmd_Lost is allowed only for the last (70th) command in the
  -- buffer.
  Cmd_Status  : INTEGER;

  -- Indicates that the new command read is not stored in the buffer (lost)
  -- because the buffer is already full.
  Lost_Cmd              : BOOLEAN := False;

  Cmd_Count             : INTEGER := 0;  -- Used as the index into Cmd_Buf.
  FIFO_Not_Empty_Count  : INTEGER := 0;  -- Running total of FIFO not empty
                                         -- condition.


  -- io status type is an array 1..16 of boolean.
  -- Spacecraft IO # 1 Status Bit Map
  --      Bit 0 - Housekeep TLM FIFO empty flag, 0 = empty
  --      Bit 1 - Housekeep TLM FIFO full  flag, 0 = empty
  --      Bit 2 - Memory Dump   FIFO empty flag, 0 = empty
  --      Bit 3 - Memory Dump   FIFO full  flag, 0 = empty
  --      Bit 4 - Universal TimeFIFO empty flag, 0 = empty
  --      Bit 5 - Universal Time Mode Status, 0=internal, 1=normal
  --      Bit 6 - Data Cmd      FIFO empty flag, 0 = empty
  --      Bit 7 - Data Cmd      FIFO full  flag, 0 = empty
  --
  IO_Status_Bits        : GLOBAL_TYPES.IO_STATUS_TYPE;
  IO_Status_Word        : GLOBAL_TYPES.WORD;


  -- Procedures and functions used in this package
--SDVS comment: CONV_WORD_TO_BIT_ARRAY is used only in parts of the code that we will not
--SDVS comment: check.
--SDVS delete:  function CONV_WORD_TO_BIT_ARRAY is new
--SDVS delete:           UNCHECKED_CONVERSION
--SDVS delete:              (SOURCE => GLOBAL_TYPES.WORD,
--SDVS delete:               TARGET => GLOBAL_TYPES.IO_STATUS_TYPE);

--SDVS comment: The function RET_MSG_ID has been added
--SDVS comment: for the scheduler. It returns the message identifier for
--SDVS comment: the next block of commands in the input encoding a message.
  function RET_MSG_ID return GLOBAL_TYPES.BYTE is --SDVS add
  begin --SDVS add
    return Cmd_Buf(1).Third_Byte; --SDVS add
  end RET_MSG_ID; --SDVS add

------------------------------------------------------------
--      FUNCTION RET_CMD_COUNT
------------------------------------------------------------
-- PURPOSE:  This function returns the command count.  The
--           command count is read to see if the command buffer
--          contains any commands to be retrieved.  Only called
--          by APP_MSGS.BUILD.
--
-- NOTES:
--
-- CHANGE HISTORY:
--   10/16/91    B. Na      Initial Version
--
-------------------------------------------------------------
  function RET_CMD_COUNT return INTEGER is

  begin

    return SERIAL_DIG_CMDS.Cmd_Count;

  end RET_CMD_COUNT;




------------------------------------------------------------
--    PROCEDURE RET_CMD_BUF_AND_STATUS
------------------------------------------------------------
-- PURPOSE : This procedure provides the commands and statuses
--           buffered to be build into application messages.
--           The commands and statuses are not double-buffered.
--           Once the buffer content is copied to the output
--           parameters, it is re-initialized.  This procedure
--           is called by APP_MSGS.BUILD.
--
-- NOTES:
--
-- CHANGE HISTORY:
--   10/16/91    B. Na      Initial Version
--
-------------------------------------------------------------
  procedure RET_CMD_BUF_AND_STATUS
           (Cmd_Buf             : out CMD_BUF_TYPE;
            Cmd_Status_Buf      : out CMD_STATUS_BUF_TYPE;
            Cmd_Count           : out INTEGER;
            FIFO_Not_Empty_Count : out INTEGER;
            IO_Status_Word      : out GLOBAL_TYPES.WORD;
            Lost_Cmd            : out BOOLEAN) is

  begin

    -- XIO RA,DSBL disables int.  Programmers should ensure that
    -- exceptions do not cause calls to Leave_Critical_Section to
    -- be missed.  Refer to page 8-21 of Tartan manual.
--SDVS delete (artclient):    Artclient.Enter_Critical_Section;

    -- Copy buffer contents and counters to output parameters.

    Cmd_Buf              := SERIAL_DIG_CMDS.Cmd_Buf;
    Cmd_Status_Buf       := SERIAL_DIG_CMDS.Cmd_Status_Buf;
    Cmd_Count            := SERIAL_DIG_CMDS.Cmd_Count;
    FIFO_Not_Empty_Count := SERIAL_DIG_CMDS.FIFO_Not_Empty_Count;
    IO_Status_Word       := SERIAL_DIG_CMDS.IO_Status_Word;
    Lost_Cmd             := SERIAL_DIG_CMDS.Lost_Cmd;

    -- Re-initialize counters.

    SERIAL_DIG_CMDS.Cmd_Count := 0;
    SERIAL_DIG_CMDS.FIFO_Not_Empty_Count := 0;
    SERIAL_DIG_CMDS.Lost_Cmd := False;

    -- XIO RA,ENBL enables int.
--SDVS delete (artclient):   Artclient.Leave_Critical_Section;

  end RET_CMD_BUF_AND_STATUS;




------------------------------------------------------------
--      PROCEDURE CMD_IN_HANDLER
------------------------------------------------------------
-- PURPOSE:  This interrupt handler is called by
--           TP_STD_INT_PROC_SERVICE_SETUP when a serial
--           digital command interrupt occurs.  This routine
--           reads the command bytes from the FIFO, checkes
--           its parity and reads and saves the spacecraft
--           IO #1 status.  Then the read commands are packed
--           into two 16-bit words and stored in the command
--           buffer.
--
--           The serial digital command FIFO address is E0800 -E09FF
--           (E1800 - E19FF ... E7800 - E79FF).  To reset the
--           Data_Cmd_FIFO, write to I/O address 0230 (0230 - 0237).
--           I/O address 8200 (8200-820f) provides status.
--           Bit D6 (numbered D0-D15) is data cmd FIFO not empty flag
--           (0 = empty).  Bit D7 is data cmd FIFO not full flag
--           ( 0 = full().  To reset the Data Cmd interrupt,
--           write to I/O address 0228 (0228 - 022F).   It does not
--           matter what the data value is.
--
-- NOTES:
--
-- CHANGE HISTORY:
--   10/16/91    B. Na      Initial Version
--
------------------------------------------------------------
  procedure CMD_IN_HANDLER is

--SDVS delete (hex):    Cmd_In_FIFO_Addr        : constant LONG_INTEGER := 16#000E0800#;
    Cmd_In_FIFO_Addr        : constant LONG_INTEGER := 919552;--SDVS replace
    -- The intrisic INPUT_WORD will add 16#8000# before performs
    -- XIO to read from the input port
```

```
--SDVS delete (hex):    IO_Status_Port          : constant INTEGER := 16#0200#;
    IO_Status_Port          : constant INTEGER := 512;--SDVS replace
--SDVS delete (hex):    Ser_Dig_Int_Port        : constant INTEGER := 16#0228#;
    Ser_Dig_Int_Port        : constant INTEGER := 552;--SDVS replace
--SDVS delete (hex):    Ser_Dig_FIFO_Reset_Port : constant INTEGER := 16#0230#;
    Ser_Dig_FIFO_Reset_Port : constant INTEGER := 560;--SDVS replace
    Data_Cmd_FIFO_Not_Empty : constant INTEGER := 6;
    Data_Cmd_FIFO_Not_Full  : constant INTEGER := 7;

    -- Operate on 4 byte (= 32 bit) serial digital command.
    Cmd_Size                : constant INTEGER := 4;

    Cmd_Unpacked   : array (1..Cmd_Size) of GLOBAL_TYPES.WORD;
    Cmd            : CMD_BYTES_TYPE;  -- 4 bytes are packed in 2 words

    Good_Parity    : BOOLEAN := TRUE;  -- indicates the parity-check
                                       -- status of Cmd.

  begin

    -- May want to move one byte at a time and check FIFO empty since
    -- parity bit is only a 50% test if do not get all 4 bytes.  However,
    -- 8085 on S/C I/O #1 always sends 4 bytes or none.  May lose some
    -- commands but should not get spurious ones.  Discuss keeping parity
    -- bit or changing it (maybe nibble parity or no parity).

    -- Read 4 bytes from cmd in FIFO into a temporary buffer.  The FIFO
    -- is only byte wide.  High order byte is not used.
    --      Cmd_Unpacked Format
    --      +--------------------+--------------------+
    --      |                    |   First Byte       |
    --      +--------------------+--------------------+
    --      |                    |   Second Byte      |
    --      +--------------------+--------------------+
    --      |                    |   Third Byte       |
    --      +--------------------+--------------------+
    --      |                    |   Fourth Byte      |
    --      +--------------------+--------------------+
--SDVS comment: Cmd_Unpacked gets a value by the call  to MEMORY_MANAGER.READ_FIFO,
--SDVS comment: which is written in 1750A. We replaced this call by the "for" loop which
--SDVS comment: gets 4 bytes from the input.
--SDVS delete (1750A):    MEMORY_MANAGER.READ_FIFO(Cmd_In_FIFO_Addr,
--SDVS delete (1750A):                             Cmd_Unpacked'address,
--SDVS delete (1750A):                             Cmd_Size);
    for i in 1 .. 4 loop --SDVS replace
       get(Cmd_Unpacked(i)); --SDVS replace
    end loop; --SDVS replace

    -- Initialize command status.
    Cmd_Status := Cmd_OK;


--SDVS comment: The next two assignment statements and the "if" statement are
--SDVS comment: used to determine the IO_Status_Word and the FIFO_Not_Empty_Count
--SDVS comment: for the command just obtained. These values are eventually passed
```

```
--SDVS comment: to APP_MSGS.RET_LATEST_CMDS which processes the information for
--SDVS comment: housekeeping telemetry. RET_LATEST_CMDS is never called in the
--SDVS comment: target software. For this reason and because the code below uses
--SDVS comment: intrinsic functions (supplied for the Tartan compiler only),
--SDVS comment: we have deleted the assignment statements and
--SDVS comment: the "if" statement. However, because the IO_Status_Word is assigned
--SDVS comment: to an out parameter of type "word" in a procedure call, SDVS requires
--SDVS comment: that, in the symbolic execution of this call, the out parameter
--SDVS comment: value be in the range 0..255. Thus, we arbitrarily assign
--SDVS comment: 0 to IO_Status_Word to meet this constraint.
    -- Read Spacecraft IO #1 Status after the data is read out of the
    -- FIFO but before the FIFO is reset or receives the next message.
    -- Refer to TP/DHS IDS, page 6.
--SDVS delete:    IO_Status_Word := INPUT_WORD(PI, IO_Status_Port);
    IO_Status_Word := 0; --SDVS add
--SDVS delete:    IO_Status_Bits := CONV_WORD_TO_BIT_ARRAY(IO_Status_Word);

--SDVS delete:    if IO_Status_Bits(Data_Cmd_FIFO_Not_Empty) then

--SDVS delete:      OUTPUT_I(16#0000#, PO, Ser_Dig_FIFO_Reset_Port);
--SDVS delete:      Cmd_Status           := Cmd_FIFO_Not_Empty;
--SDVS delete:      FIFO_Not_Empty_Count := (FIFO_Not_Empty_Count + 1) mod 256;

--SDVS delete:    end if;

    -- Check for odd parity

--SDVS delete:    Good_Parity := ASM_UTILITY.CHECK_PARITY(Cmd_Unpacked'address);
    Good_Parity := ASM_UTILITY.CHECK_PARITY;--SDVS replace
    if Good_Parity then
      Cmd_Status := Cmd_Status + Cmd_OK;
    else
      Cmd_Status := Cmd_Status + Cmd_Bad_Parity;
    end if;

    -- Packed Format
    --      +--------------------+--------------------+
    --      |  First Byte        |  Second Byte       |
    --      +--------------------+--------------------+
    --      |  Third Byte        |  Fourth Byte       |
    --      +--------------------+--------------------+
    Cmd.First_Byte  := GLOBAL_TYPES.BYTE(Cmd_Unpacked(1));
    Cmd.Second_Byte := GLOBAL_TYPES.BYTE(Cmd_Unpacked(2));
    Cmd.Third_Byte  := GLOBAL_TYPES.BYTE(Cmd_Unpacked(3));
    Cmd.Fourth_Byte := GLOBAL_TYPES.BYTE(Cmd_Unpacked(4));

    -- Reset Serial Digital Command Interrupt.
--SDVS delete (intrinsics):    OUTPUT_I(16#0000#, PO, Ser_Dig_Int_Port);

    -- Store the cmd in the buffer if the buffer is not full

    if Cmd_Count < Cmd_Buf_Size then

      -- Cannot allow reading of cmd_count till buffers are updated.
      -- Also cannot allow reading of buffers while not in a steady
      -- state.  Since done at interrupt level this is not necessary.
      -- Artclient.Enter_Critical_Section;

      -- Never wraps around because of RET_CMD_BUF_AND_STATUS proc.
      -- don't need head index - always 1.
      -- Cmd_Count is same as tail_index

      Cmd_Count := Cmd_Count + 1;

      Cmd_Buf(Cmd_Count)        := Cmd;
      Cmd_Status_Buf(Cmd_Count) := Cmd_Status;

      -- Since done at interrupt level this is not necessary.
      -- Artclient.Leave_Critical_Section;

    else

      Cmd_Status_Buf(Cmd_Buf_Size) := Cmd_Status_Buf(Cmd_Buf_Size) + Cmd_Lost;
      Lost_Cmd := True; -- This condition will be reported as system error
                        -- by APP_MSGS.BUILD.
    end if;

  end CMD_IN_HANDLER;



begin

  null;  -- There will be no specific initialization done at the elaboration
         -- of Serial_Dig_Cmds and no exception is anticipated for this
         -- package.

end SERIAL_DIG_CMDS;


--SDVS comment: JHU/APL did not give us a body for the MODE_STATE package,
--SDVS comment: since it did not appear we would need it.
--SDVS comment: These procedures allow us to write to EEPROM.
--SDVS comment: We have provided trivial bodies for them.
package body MODE_STATE is --SDVS add

 procedure PERFORM_SOFT_RESET is --SDVS add
   begin --SDVS add
     null; --SDVS add
 end PERFORM_SOFT_RESET; --SDVS add

 function RETRIEVE_CUR_STATE return MODE_STATE_TYPE is --SDVS add
   begin --SDVS add
     return Init_EEPROM_Write; --SDVS add
 end RETRIEVE_CUR_STATE; --SDVS add

 procedure CHG_STATE(Mode_State_Req    : in  MODE_STATE_TYPE; --SDVS add
                     Status_Of_Request : out STATUS_OF_REQUEST_TYPE) is --SDVS add
   begin --SDVS add
```

```
     Status_Of_Request := State_Granted; --SDVS add
   end CHG_STATE; --SDVS add

end MODE_STATE; --SDVS add

-- $users:[na.adacode.bld2.longload.ds]eeprom_data_struc.adb
------------------------------------------------------------
--          EEPROM_DATA.ADB
------------------------------------------------------------
-- TYPE: Package Body
--
-- PURPOSE : This package stores and retrieves EEPROM data.
--           There are three types of EEPROM data stored and
--           retrieved by this package:
--
--                1.  data structure
--                2.  specific memory (EEPROM) load data
--                3.  miscellaneous system variable data
--
--           The hierarchy of subprograms in this package is
--           as follows:
--
--           I.  STORE_DATA_STRUC_IN_EEPROM (called by APP_MSGS.PROCESS_MSG
--                ARRAY_OF_BLOCKS.PROCESS_MSGS_IN_BLOCKS)
--
--               A.  COPY_POINTERS_FROM_EEPROM (updates Data_Struc_Ptr_List
--                   and Free_Area pointers)
--                   1.  MEMORY_MANAGER.MOVE_FROM_PHY_LOC_TO_VARIABLE
--               B.  DETERMINE_STORAGE_LOC
--                   1.  TM_DATA.MANAGE.REPORT_SYSTEM_ERROR
--                   2.  CORRECT_BAD_STRUC_PTR
--                       2.1.  TM_DATA.MANAGE.REPORT_SYSTEM.ERROR
--                       2.2.  MEMORY_MANAGER.WRITE_BLOCK_TO_EEPROM
--                       2.3.  MEMORY_MANAGER.CONFIRM_EEPROM_DATA
--               C.  WRITE_LOAD_DATA_TO_EEPROM
--                   1.  WRITE_BLOCK
--                       1.1.  MEMORY_MANAGER.WRITE_BLOCK_TO_EEPROM
--                       1.2.  MEMORY_MANAGER.CONFIRM_EEPROM_DATA
--                       1.3.  TM_DATA.MANAGE.REPORT_SYSTEM_ERROR
--                   2.  MEMORY_MANAGER.WRITE_TO_EEPROM_BLOCK_ALIGN
--                   3.  MEMORY_MANAGER.CONFIRM_EEPROM_DATA
--                   4.  TM_DATA.MANAGE.REPORT_SYSTEM_ERROR
--               D.  ALLOC_NEW_STORAGE_AREA
--                   1.  ALLOCATE_STORAGE
--                       1.1.  UPDATE_EEPROM_STRUC_PTRS
--                             1.1.1.  MEMORY_MANAGER.WRITE_BLOCK_TO_EEPROM
--                       1.2.  UPDATE_EEPROM_FREE_PTRS
--                             1.1.1.  MEMORY_MANAGER.WRITE_BLOCK_TO_EEPROM
--                             1.1.2.  MEMORY_MANAGER.CONFIRM_EEPROM_DATA
--                   2.  CORRECT_BAD_FREE_AREA_PTR
--                       2.1  TM_DATA.MANAGE.REPORT_SYSTEM_ERROR
--                   3.  UPDATE_EEPROM_STRUC_PTRS
--                       3.1  MEMORY_MANAGER.WRITE_BLOCK_TO_EEPROM
--                   4.  TM_DATA.MANAGE.REPORT_SYSTEM_ERROR
--               E.  MEMORY_MANAGER.CONV_TYPE
--                  F.  TM_DATA.MANAGE.REPORT_SYSTEM_ERROR
--
--              II.  RET_DATA_STRUC_IN_EEPROM (called by TRKING_PARAMS.INIT_PARMS)
--
--                  A.  COPY_POINTERS_FROM_EEPROM
--                      1.  MEMORY_MANAGER.MOVE_FROM_PHY_LOC_TO_VARIABLE
--                  B.  DETERMINE_STORAGE_LOC
--                      1.  TM_DATA.MANAGE.REPORT_SYSTEM_ERROR
--                      2.  MEMORY_MANAGER.WRITE_BLOCK_TO_EEPROM
--                      3.  MEMORY_MANAGER.CONFIRM_EEPROM_DATA
--
--              III.  RET_SYS_CONFIG (called by APP_MSGS.PROCESS_MSGS_IN_BLOCKS)
--
--                  A.  MEMORY_MANAGER.MOVE_FROM_PHY_LOC_TO_VARIABLE
--                  B.  MEMORY_MANAGER.WRITE_BLOCK_TO_EEPROM
--                  C.  MEMORY_MANAGER.CONFIRM_EEPROM_DATA
--                  D.  TM_DATA.MANAGE.REPORT_SYSTEM_ERROR
--
--              IV.  WRITE_LOAD_DATA_TO_EEPROM (called bye APP_MSGS.PROCESS_
--                   MSG_TYPE and ARRAY_OF_BLOCKS.WRITE_TO_MEMORY)
--
--                  A.  WRITE_BLOCK
--                      1.  MEMORY_MANAGER.WRITE_BLOCK_TO_EEPROM
--                      2.  MEMORY_MANAGER.CONFIRM_EEPROM_DATA
--                      3.  TM_DATA.MANAGE.REPORT_SYSTEM_ERROR
--                  B.  MEMORY_MANAGER.WRITE_TO_EEPROM_BLOCK_ALIGN
--                  C.  MEMORY_MANAGER.CONFIRM_EEPROM_DATA
--                  D.  TM_DATA.MANAGE.REPORT_SYSTEM_ERROR
--
-- EXCEPTIONS:
--
-- NOTES:
--
-- CHANGE HISTORY:  01/31/92    S. HUTTON/B. Na      Initial Version
--
------------------------------------------------------------

package body EEPROM_DATA is

------------------------------------------------------------
--          PROCEDURE STORE_DATA_STRUC_IN_EEPROM
------------------------------------------------------------
-- PURPOSE : This procedure is the main control software for
--           storing data structures in EEPROM.
--
--           Before fly, in general (unless EEPROM failure location
--           before fly) Data_Struc_Ptr_List for 1st data struc block
--           are allocated  and default values exist for structures.
--           The 2nd data structure block is set as shown below to
--           designate never allocated.
--
--           Data_Struc_Ptr_List(1,Data_Struc_ID) := 68;
--           Data_Struc_Ptr_List(2,Data_Struc_ID) := 18;
--           Data_Struc_Ptr_List(3,Data_Struc_ID) := 9;
--
```

```
--           EEPROM Data Structure Block Format
--
--           Physical Address
--           (Blk 1) (Blk 2)
--           80000    BF800 +-----------------------------------------+
--                          |                                         |
--                          |                                         |
--                          |   Data_Struc_Ptr_List(1,1..79)          |
--                          |                                         |
--                          |                                         |
--           80050   BF850  +-----------------------------------------+
--                          |                                         |
--                          |                                         |
--                          |   Data_Struc_Ptr_List(2,1..79)          |
--                          |                                         |
--                          |                                         |
--           8009E   BF89E  +-----------------------------------------+
--                          |                                         |
--                          |                                         |
--                          |   Data_Struc_Ptr_List(3,1..79)          |
--                          |                                         |
--                          |                                         |
--           800ED   BF8ED  +-----------------------------------------+
--                          |        Free_Area pointer (1)            |
--                          |        Free_Area pointer (2)            |
--                          |        Free_Area pointer (3)            |
--           800F0   BF8F0  +-----------------------------------------+
--                          |                                         |
--                          |        Data Structure Storage           |
--                          |               Area                      |
--                          |                                         |
--                          |     -----------------------------       |
--                          |                                         |
--                          |                                         |
--                          |                                         |
--                          |        Data Stucture Storage            |
--                          |              Free Area                  |
--                          |                                         |
--           807FF   BFFFF  |                                         |
--                          +-----------------------------------------+
--
--
--
-- NOTES:
--
-- CHANGE HISTORY:  01/10/92    S. HUTTON/B. NA      Original
--
-----------------------------------------------------------
  procedure STORE_DATA_STRUC_IN_EEPROM
       (Array_Of_Words          : in      CMDS_TYPES.APP_MSG_OUT_TYPE;
        Data_Struc_ID           : in      INTEGER;
        Data_Struc_Length       : in      INTEGER) is


   use INTRINSIC_FUNCTIONS; --SDVS add
  begin

     for I in 2 .. Data_Struc_Length + 1 loop --SDVS replace
       put(Array_Of_Words(i)); --SDVS replace
     end loop; --SDVS replace


  end STORE_DATA_STRUC_IN_EEPROM;




begin

  null;

end EEPROM_DATA;




--SDVS comment: The package body of TM_DATA below is only a simplification
--SDVS comment: of the real TM_DATA body. In the real one, MANAGE is a task
--SDVS comment: and REPORT_SYSTEM_ERROR is one of its entries. Furthermore,
--SDVS comment: we do not even have the body of TM_DATA and it probably does
--SDVS comment: not contain a "put." We dealt with these problems in the manner
--SDVS comment: below. No branch in the SDVS execution lead to a call of
--SDVS comment: the procedure REPORT_SYSTEM_ERROR
package body TM_DATA is

  package body MANAGE is --SDVS replacement

    procedure REPORT_SYSTEM_ERROR(Sys_Error_Code :
                                     in GLOBAL_TYPES.BYTE) is --SDVS replacement
      begin --SDVS replacement
      put(Sys_Error_Code); --SDVS replacement
    end  REPORT_SYSTEM_ERROR; --SDVS replacement

  end MANAGE; --SDVS replacement

end TM_DATA;

-- $users:[na.adacode.bld2.longload]app_msgs.adb
--------------------------------------------------------------------------------
--          APP_MSGS.ADB
--------------------------------------------------------------------------------
-- TYPE: Package Body
--
-- PURPOSE : This package contains software to build and
--           process the application messages for the serial digital
--          commands buffered by SERIAL_DIG_CMDS.
--
--    The hierachy of subprograms in this package is as follows:
--
--    I.    BUILD  (This task runs at a 2 Hz rate and initiates a rendezvous
--                  with MANAGE_MSG_RETRIEVAL at the entry point
--                  NOTIFY_MSG_STORED.)
--          A.  SERIAL_DIG_CMDS.RET_CMD_COUNT
--          B.  SERIAL_DIG_CMDS.RET_CMD_BUF_AND_STATUS
--          C.  TM_DATA.MANAGE.REPORT_SYSTEM_ERROR
--          D.  INITIATE_BUILD
--          E.  CONTINUE_BUILD
--          F.  MANAGE_MSG_RETRIEVAL.NOTIFY_MSG_STORED
--
--    II.   PROCESS_MSG  (This task runs without any delay but pends at the
--                        guarded entry MANAGE_MSG_RETRIEVAL.RET_NEXT_MSG.)
--          A.  MANAGE_MSG_RETRIEVAL.RET_NEXT_MSG
--          B.  MODE_STATE.RETRIEVE_CUR_STATE
--          C.  MODE_STATE.CHG_STATE
--          D.  EEPROM_DATA.STORE_DATA_STRUC_IN_EEPROM
--          E.  TM_DATA.MANAGE.REPORT_SYSTEM_ERROR
--          F.  TRKING_PARAMS.MANAGE.STORE_TRKING_STRUC
--          G.  MEMORY_MANAGER.WRITE_BLOCK_TO_EEPROM
--          H.  PRIME_SCIENCE_DATA.SET_PRIME_SCI_DATA_RATE
--          I.  MODE_STATE.PERFORM_SOFT_RESET
--          J.  ARRAY_OF_BLOCKS.INIT_LONG_LOAD
--          K.  ARRAY_OF_BLOCKS.PROCESS_MSGS_IN_BLOCKS.PREPARE_FOR_TIMEOUT
--          L.  MEMORY_MANAGER.CONV_TYPE
--          M.  POINTING_INFO_OUT.SHUTDOWN_IMMINENT
--          N.  TM.PROCESS_CMD.UPDATE_VAR_MON_ADDR
--          O.  TM_PROCESS_CMD.CHK_NO_OP_CMD
--          P.  EEPROM_DATA.WRITE_LOAD_DATA_TO_EEPROM
--
--    III.  MANAGE_MSG_RETRIEVAL (This task rendezvous with PROCESS_MSG when
--                                the application message count is greater
--                                than zero.  This task also rendezvous with
--                                BUILD when the application message queue
--                                becomes not empty from empty.)
--          A.  MEMORY_MANAGER.CONV_TYPE
--
--    IV.   Other Procedures or Functions
--            A.  NEW_TM_DATA
--            B.  RET_LATEST_CMDS
--            C.  RET_INIT_CMD_LAST_APP_MSG
--            D.  INITIATE_BUILD
--                1.  TM_DATA.MANAGE.REPORT_SYSTEM_ERROR
--            E.  CONTINUE_BUILD
--                1.  TM_DATA.MANAGE.REPORT_SYSTEM_ERROR
--
-- EXCEPTIONS:
--
-- NOTES:
--
-- CHANGE HISTORY:
--    10/16/91    B. Na      Initial Version
--   01/31/92    B. Na      Build 2 Changes
--
------------------------------------------------------------

--SDVS comment: For the body of APP_MSGS, we delete all "with" and "use"
--SDVS comment: clauses appearing before its declarative part. However, we
--SDVS comment: move all "use" clauses for packages that we actually need
--SDVS comment: within and in the beginning of the declarative part
--SDVS comment: of the  APP_MSGS body.

--SDVS delete (with): with ARTCLIENT;
--SDVS delete (with): with SYSTEM; use SYSTEM;
--SDVS delete (with): with INTRINSIC_FUNCTIONS; use INTRINSIC_FUNCTIONS;
--SDVS delete (with): with UNCHECKED_CONVERSION;

--SDVS delete (with): with GLOBAL_TYPES; use GLOBAL_TYPES;  -- Globally used types.
--SDVS delete (with): with CMDS_TYPES;  -- Commonly used types among the command systems
                  -- group packages.
--SDVS delete (with): with TRKING_DATA_STRUC;  -- contains number of words per
-- data structure table

--with SYSTEM_TIME;
--SDVS delete (with): with MEMORY_MANAGER;
-- 4/21/92 changed sys_error_type from enum to integer.
--SDVS delete (with): with TM_DATA; -- no longer need use TM_DATA;
--SDVS delete (with): with TM; use TM;
--SDVS delete (with): with MODE_STATE; use MODE_STATE;
--SDVS delete (with): with SERIAL_DIG_CMDS; use SERIAL_DIG_CMDS;
--SDVS delete (with): with PRIME_SCIENCE_DATA;
--SDVS delete (with): with TRKING_PARAMS;
--SDVS delete (with): with POINTING_INFO_OUT;
--SDVS delete (with): with ARRAY_OF_BLOCKS;
--SDVS delete (with): with EEPROM_DATA;

-- REMOVE LATER!!!
--SDVS delete (with): with SIMPLE_IO; use SIMPLE_IO;

--SDVS delete (pragma): pragma elaborate (TM_DATA);
--SDVS delete (pragma): pragma elaborate (TRKING_PARAMS);
--SDVS delete (pragma): pragma elaborate (EEPROM_DATA);
--SDVS delete (pragma): pragma elaborate (INTRINSIC_FUNCTIONS);
--SDVS delete (pragma): pragma elaborate (PRIME_SCIENCE_DATA);

package body APP_MSGS is

use INTRINSIC_FUNCTIONS; --SDVS replace (reposition)
use GLOBAL_TYPES;  --SDVS replace (reposition)
--SDVS delete use TM; --SDVS replace (reposition)
use MODE_STATE; --SDVS replace (reposition)
use SERIAL_DIG_CMDS; --SDVS replace (reposition)

  -- I.  Variables Associated With Retrieved Commands From SERIAL_DIG_CMDS.

  -- Holds the number of commands retrieved.
  Cmd_Count               : INTEGER;
  -- Retrieved commands are stored in this buffer.
  Cmd_Buf                 : SERIAL_DIG_CMDS.CMD_BUF_TYPE;
  -- Retrieved command statuses are stored in this buffer.
  Cmd_Status_Buf          : SERIAL_DIG_CMDS.CMD_STATUS_BUF_TYPE;

  -- The following defines the possible command status values with the
  -- two exceptions:
  --        (1) Cmd_Lost is allowed only for the last (70th) command
  --            in the buffer.
  --        (2) Cmd_Lost and Cmd_FIFO_Not_Empty may be ORed with the rest
  --            of status values.
  --
  -- Cmd_OK                      = 0
  -- App_Msg_Dropped             = 1
  -- Bad_Checksum                = 2
  -- Flushed_prior_Build         = 3
  -- Invalid_Data_Struct_Msg_ID = 4
  -- Invalid_Starting_Cmd        = 5
  -- App_Msg_Overwritten         = 6
  -- (The following three conditions are checked by SERIAL_DIG_CMDS)
  -- Cmd_Bad_Parity              = 7  (Parity of the command is not odd)
  -- Cmd_Lost                    = 8  (New command is lost because the
  --                                   command buffer is full)
  -- Cmd_FIFO_Not_Empty          = 16 (The command-in FIFO is not empty after
  --                                   one command is read)

  -- This flag indicates that one or more commands read from the FIFO were
  -- not stored in the buffer because the buffer was full.  This flag is
  -- checked by APP_MSGS.BUILD to generate the system error message.
  Lost_Cmd                : BOOLEAN := False;
  -- Holds the number of occurances that FIFO is not empty after one command
  -- (4 bytes) is read.
  FIFO_Not_Empty_Count    : INTEGER := 0;

  Rejected_Cmd_Count      : INTEGER := 0;  -- 0 thru 255
  Max_Rejected_Cmd_Count : constant INTEGER := 256;


  -- II. Variables Set and Used While Building Message

  -- The messages built and ready to be retrieved and processed are stored
  -- in this queue.  4/21/92 changed App_Msg_Q_Size from = 30 to = 15 to
  -- save space; saves 15 * 80 = 1200 words. changed back to =30 5/15/92
  App_Msg_Q_Size          : constant INTEGER := 30;

  type APP_MSG_TYPE is array (1..CMDS_TYPES.App_Msg_Max) of GLOBAL_TYPES.WORD;
--SDVS delete (2-dimensional array):  type APP_MSG_Q_TYPE is array
--SDVS delete (2-dimensional array): (1..App_Msg_Q_Size, 1..CMDS_TYPES.App_Msg_Max)
--SDVS delete (2-dimensional array):   of GLOBAL_TYPES.WORD;
  type APP_MSG_Q_TYPE is array (1..App_Msg_Q_Size) of APP_MSG_TYPE; --SDVS replace
  App_Msg_Q               : APP_MSG_Q_TYPE;

  -- This buffer holds the intermediate result while the message is being
  -- built.  As the message become complete, the completed message is moved
  -- to App-Msg-Q.
```

```
--SDVS delete (others):  App_Msg               : APP_MSG_TYPE := (others => 0);
  App_Msg                 : APP_MSG_TYPE; --SDVS replace

  -- Used to control build process.  A logic False indicates the start
  -- of build process.  Will be set to True by INITIATE_BUILD when
  -- more commands are needed to build the message.  Any one of the
  -- following abnomalies will set this variable to False:
  --      Cmd parity fail
  --      Msg checksum fail
  --      Invalid op-code
  Build_In_Progress       : BOOLEAN := False;

  -- First byte (opcode and parity) of the start command of the multiple
  -- commands message is saved; so it can be reported to TM as the start
  -- command associated with the last received application message.
  Starting_Cmd_With_Par   : GLOBAL_TYPES.BYTE;

  -- Used as a word index into the App_Msg while build is in progress to
  -- store incoming command bytes.
  Word_Cntr               : INTEGER := 0;

  -- Holds the number of commands received for the message currently
  -- being built.
  Cmds_Rcvd               : INTEGER := 0;

  -- Set by Initiate_Build and used by Continue_Build.  Holds the number
  -- of required commands to be recevied for the build.
  Num_Cmds_For_App_Msg    : INTEGER := 0;

  -- Set by Initiate_Build and used by Continue_Build.  Holds the the number
  -- of words of the message to be stored into the application message queue.
  Num_Words_For_App_Msg   : INTEGER := 0;

  -- Set by Initiate_Build and used by Continue_Build.  Holds the next expected
  -- command type to complete the build.
  Cont_Cmd                : GLOBAL_TYPES.BYTE;

  -- Set and used only by Initiate_Build and Continue_Build.  Holds the
  -- last written position in the application message queue.
  Q_Tail                  : INTEGER := 0;

  -- Set and used only by MANAGE_MSG_RETRIEVAL.  Holds the next
  -- read position in the application message queue.
  Q_Head                  : INTEGER := 1;

  -- Holds the number of messages built and stored in the queue.
  App_Msg_Counter         : INTEGER := 0;

  -- Holds the start command associated with the last received message.
--SDVS delete (others):  Cmd_Last_App_Msg
--SDVS delete (others): : SERIAL_DIG_CMDS.CMD_BYTES_TYPE := (others => 0);
  Cmd_Last_App_Msg        : SERIAL_DIG_CMDS.CMD_BYTES_TYPE; --SDVS replace

  -- III.  Starting Command Op-Codes (and Continuation Codes)
  --
  -- +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  -- | P |       OP-CODE             |   Cmd Data (Bits 0 - 7)       |
  -- +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  -- |                            Cmd Data (Bits 8 - 23)             |
  -- +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  Data_Struc_Load        : constant GLOBAL_TYPES.BYTE :=  1;
  RAM_Mem_Load           : constant GLOBAL_TYPES.BYTE :=  2;
  EEPROM_Mem_Load        : constant GLOBAL_TYPES.BYTE :=  4;
  Set_RAM_Def_Tab_Entry  : constant GLOBAL_TYPES.BYTE :=  7;
  Prep_For_Long_Load     : constant GLOBAL_TYPES.BYTE := 11;
  Mem_Dump_X_Frames      : constant GLOBAL_TYPES.BYTE := 13;
  Shutdown_Imminent      : constant GLOBAL_TYPES.BYTE := 19;
  Sat_Subsys_Config      : constant GLOBAL_TYPES.BYTE := 22;
  No_op_Cmd              : constant GLOBAL_TYPES.BYTE := 25;
  Chg_Monitor_Loc        : constant GLOBAL_TYPES.BYTE := 28;
  Soft_Reset             : constant GLOBAL_TYPES.BYTE := 32;
  Mem_Dump_Data_Struc    : constant GLOBAL_TYPES.BYTE := 35;
  Clr_Timeout_Telltale   : constant GLOBAL_TYPES.BYTE := 37;
  UT_Control             : constant GLOBAL_TYPES.BYTE := 38;

  -- Continuation commands
  Data_Load_Cmd          : constant GLOBAL_TYPES.BYTE :=  8;
  Ext_Dump_X_Frames      : constant GLOBAL_TYPES.BYTE := 16;
  Ext_Chg_Mon_Loc        : constant GLOBAL_TYPES.BYTE := 31;




  -- IV.  Variables Used To Buffer and Report Last 40 Commands To TM.

--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS
--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS
--SDVS    The following group of declarations are used to report     SDVS
--SDVS    to telemetry. The declared objects are not used in the     SDVS
--SDVS    target code and are therefore deleted. Any assignments     SDVS
--SDVS    to them in BUILD are also deleted.                         SDVS
--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS
--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS

  -- Used in RET_LATEST_CMDS to determine the first command in
  -- Latest_Cmds_Q to be reported to TM at this frame.
--SDVS delete:  Read_Ptr              : INTEGER := 0;

  -- Points to the last written entry in the queue.  Queue entries
  -- extend from 1 to 75.  Incremented by BUILD as the incoming
  -- commands are buffered in the Latest_Cmds_Q
--SDVS delete:  Write_Ptr             : INTEGER := 0;

--SDVS delete:  Latest_Cmds_Q_Size    : constant INTEGER := 75;

  -- Holds up to 75 commands and statuses which will be reported to TM.

--SDVS delete:  Zero_Cmd              : constant Serial_Dig_Cmds.Cmd_Bytes_Type
```

```
--SDVS delete:                              := (0,0,0,0);

--SDVS delete:  Latest_Cmds_Q          : array (1..Latest_Cmds_Q_Size) of
--SDVS delete:                              Serial_Dig_Cmds.Cmd_Bytes_Type;
--SDVS delete:                              := (others => (0,0,0,0));

--SDVS delete:  Latest_Status_Q        : array (1..Latest_Cmds_Q_Size) of INTEGER :=
--SDVS delete:                              (others => Cmd_OK);
--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS
--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS

  -- IV.  Constants Associated With Data Structures.

  -- These data structures are used only by INITIATE_BUILD.  However,
  -- to conserve runtime stack space, the structures are defined here
  -- rather than in INITIATE_BUILD.

  -- This table lists the number of commands required to build
  -- a data structure load message.  The data structure ID is used as an
  -- index into the table.  ID 1 thru 49 are reserved for current/future
  -- event, 50 thru 79 are reserved for next event and 80 thru 109 are
  -- are reserved for current data structures.

  type CMDS_FOR_DATA_STRUC_TYPE is array (1..TRKING_DATA_STRUC.Num_Data_Struc)
                                  of integer;
  Cmds_For_Data_Struc : constant CMDS_FOR_DATA_STRUC_TYPE :=
      (14, 14, 14, 14, 14, 14, 14, 10, 10, 03,
       03, 04, 04, 03, 03, 03, 03, 03, 08, 03,
       03, 02, 04, 02, 04, 04, 04, 02, 06, 00,
       00, 00, 00, 00, 00, 00, 00, 00, 00, 00,
       00, 00, 00, 00, 00, 00, 00, 00, 00,

       10, 04, 06, 38, 38, 52, 52, 13, 06, 04,
       06, 04, 05, 11, 07, 08, 08, 15, 00, 00,
       00, 00, 00, 00, 00, 00, 00, 00, 00, 00,

       04, 02, 02, 02, 02, 02, 04, 04, 04, 04,
       02, 11, 07, 02, 00, 00, 00, 00, 00, 00,
       00, 00, 00, 00, 00, 00, 00, 00, 00, 00);


  -- IV.  Other Micellaneous Variables.

  -- Last commanded satellite subsystem configuration value saved
  -- to be reported to TM.  If the "Store Subsys Config in EEPROM"
  -- bit is set, then the value will be stored in EEPROM.
  Saved_Sat_Subsys_Config : GLOBAL_TYPES.WORD := 0;

  -- Holds the Data structure to be stored in EEPROM or memory load
  -- data to be stored in EEPROM.  This array is passed to the
  -- EEPROM_DATA.WRITE_LOAD_DATA_TO_EEPROM to modify EEPROM.
  Load_Msg                : CMDS_TYPES.LOAD_MSG_TYPE;

  -- The range of Cmd_Total will be 0 to 31999.
  Cmd_Total      : INTEGER;
```

```
   Cmd_Total_Max  : constant INTEGER := 32000;

   -- Holds the spacecraft IO #1 status to be reported in TM.
   IO_Status_Word : GLOBAL_TYPES.WORD;


   --  Tasks, Procedures and Functions Used

   function CONV_WORD_TO_INTEGER      is new UNCHECKED_CONVERSION
                                              (SOURCE => GLOBAL_TYPES.WORD,
                                               TARGET => INTEGER);

   function CONV_BYTE_TO_INTEGER      is new UNCHECKED_CONVERSION
                                              (SOURCE => GLOBAL_TYPES.BYTE,
                                               TARGET => INTEGER);

   function CONV_INTEGER_TO_BYTE      is new UNCHECKED_CONVERSION
                                              (SOURCE => INTEGER,
                                               TARGET => GLOBAL_TYPES.BYTE);

   function CONV_BYTE_TO_WORD         is new UNCHECKED_CONVERSION
                                              (SOURCE => GLOBAL_TYPES.BYTE,
                                               TARGET => GLOBAL_TYPES.WORD);

   function CONV_WORD_TO_BYTE         is new UNCHECKED_CONVERSION
                                              (SOURCE => GLOBAL_TYPES.WORD,
                                               TARGET => GLOBAL_TYPES.BYTE);

   procedure INITIATE_BUILD (Start_Op_Code : in GLOBAL_TYPES.BYTE;
                             Cmd_Buf_Index : in INTEGER);

   procedure CONTINUE_BUILD (Cont_Op_Code  : in GLOBAL_TYPES.BYTE;
                             Cmd_Buf_Index : in INTEGER);



--SDVS delete (task):  task type BUILD_TYPE is
--SDVS delete (task):    pragma priority (18);
--SDVS delete (task):  end BUILD_TYPE;
--SDVS delete (task):  for BUILD_TYPE'Storage_Size use 350;
--SDVS delete (task):  BUILD : BUILD_TYPE;


--SDVS delete (task):  task type PROCESS_MSG_TYPE is
--SDVS delete (task):    pragma priority (13);
--SDVS delete (task):  end PROCESS_MSG_TYPE;
--SDVS delete (task):--  for PROCESS_MSG_TYPE'Storage_Size use 4000;
--SDVS delete (task):  for PROCESS_MSG_TYPE'Storage_Size use 700;
--SDVS delete (task):  PROCESS_MSG : PROCESS_MSG_TYPE;


--SDVS delete (task):  task type MANAGE_MSG_RETRIEVAL_TYPE is
--SDVS delete (task):    entry RET_NEXT_MSG (App_Msg_Out : out CMDS_TYPES.APP_MSG_OUT_TYPE);
--SDVS delete (task):    entry NOTIFY_MSG_STORED;
--SDVS delete (task):    pragma priority (14);
```

```
--SDVS delete (task):  end MANAGE_MSG_RETRIEVAL_TYPE;
--SDVS delete (task):  for MANAGE_MSG_RETRIEVAL_TYPE'storage_size use 150;
--SDVS delete (task):  MANAGE_MSG_RETRIEVAL : MANAGE_MSG_RETRIEVAL_TYPE;



-----------------------------------------------------------------------------
--      TASK BUILD
-----------------------------------------------------------------------------
-- PURPOSE :  This task first checks if there is any commands
--            to be processed by calling SERIAL_DIG_CMDS.  If no command
--            is available to be processed, then the task 'sleeps' for
--            two seconds and repeats the task.
--
--            If there are commands to be processed, then the cmd buffer,
--            status buffer and the other associated variables are retrieved
--            from SERIAL_DIG_CMDS.  For each command in the buffer, its status
--            is checked for errors such as parity error and FIFO not
--            empty and command lost errors.  Then, either CONTINUE_BUILD
--            or INITIATE_BUILD is called to build the application message(s).
--            out of commands.
--
--            First the type of application to be built is determined.
--            If the application message has a checksum, then the message is
--            built in the form to validate the checksum.  Once the checksum
--            is verified, the application message is stored in the queue of
--            application messages.
--
--            A queue of the last 75 commands is also maintained to be
--            collected by TM.COLLECT.
--
--            This task does not have any entry points.
--
--
-- NOTES:
--
-- CHANGE HISTORY:
--   10/16/91    B. Na      Initial Version
--
-----------------------------------------------------------------------------
--SDVS delete (task):  task body BUILD_TYPE is
procedure BUILD is --SDVS replace

--SDVS comment: The four declarations that follow are never used in the code.
--SDVS comment: Actually, Time_To_Run is set to 0 but never used thereafter.
--SDVS delete:   Delay_Time                : LONG_FLOAT;
--SDVS delete:   Time_To_Run               : LONG_FLOAT;
--SDVS delete:   Current_UT_Time           : LONG_FLOAT;
--SDVS delete:   Time_Interv_Between_Exec : constant LONG_FLOAT := 0.5; --

    Old_App_Msg_Count        : INTEGER;
    Cmd_Buf_Index            : INTEGER;

    Op_Code                  : GLOBAL_TYPES.BYTE;
```

```
--SDVS delete (hex):    Parity_Bit_Mask          : constant GLOBAL_TYPES.BYTE := 16#7F#;
    Parity_Bit_Mask          : constant GLOBAL_TYPES.BYTE := 127; --SDVS replace

    Saved_Cmd_Status         : INTEGER;

  begin

    -- Mark the task start time to be used in calculating the delay
    -- duration required to schedule this task for a 2 Hz rate.

    -- Time_To_Run := SYSTEM_TIME.RETURN_CUR_TIME;
--SDVS delete:    Time_To_Run := 0.0;

--SDVS delete (task loop):    loop

--SDVS delete (task loop):      loop -- for tasking_error
--SDVS delete (task loop):      begin

        -- Watchdog timer is initially armed in the boot program just
        -- before handing over control to the application program.  The
        -- application program has 65 ms to stop or reset the timer.
        -- The application program then rearm the timer for TBD seconds.
        -- This task periodically reset the timer to prevent the reset
        -- of the tracking processor.

--SDVS delete (intrinsics):        OUTPUT_WORD(16#0000#, GO, 0);



      -- Check if any serial digital command is received and buffered.
      Cmd_Count := SERIAL_DIG_CMDS.RET_CMD_COUNT;

      if Cmd_Count > 0 then

         -- Retrieve the buffered commands.
         SERIAL_DIG_CMDS.RET_CMD_BUF_AND_STATUS
                  (Cmd_Buf,
                   Cmd_Status_Buf,
                   Cmd_Count,
                   FIFO_Not_Empty_Count,
                   IO_Status_Word,
                   Lost_Cmd);

        -- total number of cmds received since power-up excluding the
        -- number of cmds received during boot.
        Cmd_Total := (Cmd_Total + Cmd_Count) mod Cmd_Total_Max;

        if Lost_Cmd then
          -- Cmd_Lost_Occured = 1.
          TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(1);
        end if;

        -- Used to notify buffer task, MANAGE_MSG_RETRIVAL, that now
        -- have messages to process.
        Old_App_Msg_Count := App_Msg_Counter;
```

```
 for Cmd_Buf_Index in 1..Cmd_Count loop

   Saved_Cmd_Status := Cmd_Status_Buf(Cmd_Buf_Index);

   -- Temporarlly mask Cmd_FIFO_Not_Empty (bit 4) and Cmd_Lost (bit 3)
   -- of the command status word during the build process.

   Cmd_Status_Buf(Cmd_Buf_Index) :=
              AND_I(Cmd_Status_Buf(Cmd_Buf_Index), 7);

   if Cmd_Status_Buf(Cmd_Buf_Index) = Cmd_OK then

     -- zero parity bit
Op_Code :=
  AND_BYTE(Cmd_Buf(Cmd_Buf_Index).First_Byte, Parity_Bit_Mask);

     if Build_In_Progress then

       -- If build complete, stores the message in queue to
       -- process and increment App_Msg_Counter.
       Continue_Build(Op_Code, Cmd_Buf_Index);

     else

       -- If build complete, stores the message in queue to
       -- process and increment App_Msg_Counter.
       Initiate_Build(Op_Code, Cmd_Buf_Index);

     end if;

   else  -- Cmd_Status_Buf(Cmd_Buf_Index) /= Cmd_OK
         -- implies bad parity
     -- Parity_Failed = 16
     TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(16);

     -- Update the number of rejected commands to report in TM.
     Rejected_Cmd_Count :=
           (Rejected_Cmd_Count + 1) mod Max_Rejected_Cmd_Count;

     -- Terminate build process
     Build_In_Progress := False;

   end if;

   if Saved_Cmd_Status > 7 then

   -- If Cmd_Lost and/or Cmd_FIFO_Not_Empty condition(s) exists,
   -- then set the corresponding bit(s) of the command status word.
   -- These bits were temporarlly masked during the build process.

     Cmd_Status_Buf(Cmd_Buf_Index) :=
         OR_I(Saved_Cmd_Status, Cmd_Status_Buf(Cmd_Buf_Index));

   end if;
```

```
        end loop; -- for Cmd_Buf_Index in 1..Cmd_Count loop



        -- APP_MSGS provides the last 40 commands received to TM.
        -- Since TM collects data once a second on average, APP_MSGS
        -- queues up to 75 commands to give TM some time to catch up
        -- in the event of heavy influx of commands in short period
        -- of time.

--SDVS comment: As we noted in the specification of APP_MSGS, the function
--SDVS comment: NEW_TM_DATA, and the two procedures RET_LATEST_CMDS, and
--SDVS comment: RET_INIT_CMD_LAST_APP_MSG are used to report to telemetry.
--SDVS comment: Since they are not used in the target code to build or
--SDVS comment: process messages (and are never called by the target software),
--SDVS comment: we deleted them. The loop that follows assigns values to
--SDVS comment: objects which are used only by NEW_TM_DATA, RET_LATEST_CMDS,
--SDVS comment: and RET_INIT_CMD_LAST_APP_MSG. Thus, we deleted the entire
--SDVS comment  loop.

--SDVS delete:       for Cmd_Buf_Index in 1..Cmd_Count loop

         -- Latest_Cmds_Q will be Q of last 75 cmds.  Cmds are stored
         -- here until these are collected, up to 40 cmds per frame,
         -- by TM.COLLECT.

--SDVS delete (artclient):         Artclient.Enter_Critical_Section;

           -- Write_Ptr is used to report the last 40 commands to TM.
           -- If, for some reason, TM.COLLECT fails to collect data
           -- as scheduled, then Write_Ptr could wrap around and
           -- pass Read_Ptr (i.e. queue overflow condition).  In this
           -- case, the commands received before the 75th command
           -- (counting backward, most recent command being 1st) will
           -- be not reported to TM.

--SDVS delete:           Write_Ptr := (Write_Ptr mod 75) + 1;

--SDVS delete:           Latest_Cmds_Q(Write_Ptr)   := Cmd_Buf(Cmd_Buf_Index);
--SDVS delete:           Latest_Status_Q(Write_Ptr) := Cmd_Status_Buf(Cmd_Buf_Index);

--SDVS delete (artclient):         Artclient.Leave_Critical_Section;

--SDVS delete:       end loop;



--SDVS delete (task):       if (Old_App_Msg_Count = 0) and
--SDVS delete (task):          (App_Msg_Counter > 0)   then

--SDVS delete (task):         MANAGE_MSG_RETRIEVAL.NOTIFY_MSG_STORED;

--SDVS delete (task):       end if;
```

```
      end if;  -- if Cmd_Count > 0 then



      --Time_To_Run := Time_To_Run + Time_Interv_Between_Exec;
      --Current_UT_Time := SYSTEM_TIME.RETURN_CUR_TIME;

      --if Time_To_Run > Current_UT_Time then
        --Delay_Time := Time_To_Run - Current_UT_Time;
        --DELAY(DURATION(Delay_Time));
--SDVS delete (task):         DELAY(0.5);
      --end if;


--SDVS delete (task):      exit;

--SDVS delete (exception):      exception

--SDVS delete (exception):        when tasking_error =>

--SDVS delete (exception):          writestring("apbte 1");
--SDVS delete (exception):         writeln;
--SDVS delete (exception):         -- App_Msgs_Build_Tasking_Exception = 86
--SDVS delete (exception):          TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(86);

--SDVS delete (exception):        when others =>

--SDVS delete (exception):         writestring("apbe 2");
--SDVS delete (exception):          writeln;
          -- App_Msgs_Build_Exception = 87
--SDVS delete (exception):          TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(87);

--SDVS delete      end; -- for begin

--SDVS delete (task loop):      end loop;

--SDVS delete (task loop):    end loop;

--SDVS delete (exception):    exception

--SDVS delete (exception):      when tasking_error =>

--SDVS delete (exception):        writestring("apbteol 3");
--SDVS delete (exception):        writeln;

--SDVS delete (exception):      when others =>

--SDVS delete (exception):        writestring("apbeol 4");
--SDVS delete (exception):        writeln;

  end BUILD;


-------------------------------------------------------------------------------
```

```
--      TASK PROCESS_MSG
-------------------------------------------------------------------------------
-- PURPOSE:  This task retrieves and processes the next
--            available application message from the application
--            message queue.

--            This task continuously initiates a rendezvous with
--            MANAGE_MSG_RETRIEVAL at the entry point RET_NEXT_MSG.
--            This task does not have any entry points.
--
-- NOTES:
--
-- CHANGE HISTORY:
--   10/16/91    B. Na      Initial Version
--   mm/dd/yy    B. Na      Build 2 Changes
--
-------------------------------------------------------------------------------

--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS
--SDVS The task  PROCESS_MSG processes 11 types of application messages.--SDVS
--SDVS Since we are only interested in data-structure messages and the  --SDVS
--SDVS messages are processed in an Ada ``case'' statement, we          --SDVS
--SDVS deleted all the statements in all of the cases (and substituted  --SDVS
--SDVS the ``null'' statement for them) except for the                 --SDVS
--SDVS Data_Struc_Load case.  Any other uncommented deletions are for   --SDVS
--SDVS assignments to objects that are used only in the deleted cases.  --SDVS
--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS


--SDVS delete (task):  task body PROCESS_MSG_TYPE is
procedure PROCESS_MSG is --SDVS replace
    -- Holds the message to be processed next.
--SDVS delete (unconstrained array , others):
--SDVS cont: New_App_Msg : CMDS_TYPES.APP_MSG_OUT_TYPE(CMDS_TYPES.INDEX_SUBTYPE)
--SDVS cont:                                      := (others => 0);
    New_App_Msg : CMDS_TYPES.APP_MSG_OUT_TYPE; --SDVS replace
    New_Cmd_Op_Code           : GLOBAL_TYPES.BYTE;

    -- Holds intermediate results while performing bitwise
    -- operations.
    Temp_Word                 : GLOBAL_TYPES.WORD;
    Temp_Word_1               : GLOBAL_TYPES.WORD;
    Temp_Word_2               : GLOBAL_TYPES.WORD;

    TP_Num_And_Config_Save_Bits : GLOBAL_TYPES.WORD;
    Subsys_Config_Bits          : GLOBAL_TYPES.WORD;

    Addr_Of_RAM_Load          : LONG_INTEGER;
    Last_Addr_Load            : LONG_INTEGER;
    Dump_Address              : LONG_INTEGER;

--SDVS delete:    Dump_Params              : TM.DUMP_PARAMS_TYPE;
--SDVS delete:    Mem_Locations_To_Monitor : TM.LOCS_TO_MON_TYPE;
    Status_Of_Request         : MODE_STATE.STATUS_OF_REQUEST_TYPE;
    Curr_Mode_State           : MODE_STATE.MODE_STATE_TYPE;
    -- Data structure storage selection values.
    Update_RAM              : constant INTEGER := 1;
    Update_EEPROM           : constant INTEGER := 2;
    Update_Both             : constant INTEGER := 3;

    UT_Internal             : constant INTEGER := 1;

    Storage_Selection       : INTEGER;

    Memory_Upload_Select    : INTEGER;

    Data_Struc_ID           : INTEGER;
    Data_Struc_Length       : INTEGER;

    -- Used to write load data or data sturcture message to EEPROM.
    Load_Msg_Is_Data_Struc  : BOOLEAN;
    Stored                  : BOOLEAN;


  begin

    -- Configure satellite subsystem with the configuration word
    -- preserved in EEPROM.

--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS
--SDVS comment: The next block of code (up to the simple loop) is executed
--SDVS only once at the elaboration of the task. It serves no function
--SDVS for our purposes.
--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS--SDVS

--SDVS delete:    Saved_Sat_Subsys_Config := EEPROM_DATA.RET_SYS_CONFIG;

--SDVS delete:    OUTPUT_WORD(Saved_Sat_Subsys_Config, PO, 16#0028#);


    -- Set prime science data rate based on Bit 4 of
    -- Saved_Sat_Subsys_Config.

--SDVS delete:    Temp_Word := AND_WORD(Saved_Sat_Subsys_Config, 16#0800#);
--SDVS delete:    if Temp_Word = 16#0800# then
--SDVS delete:      PRIME_SCIENCE_DATA.SET_PRIME_SCI_DATA_RATE(5);
--SDVS delete:    else
--SDVS delete:      PRIME_SCIENCE_DATA.SET_PRIME_SCI_DATA_RATE(25);
--SDVS delete:    end if;


--SDVS delete (task loop):    loop

--SDVS delete (task loop):      loop -- for tasking_error
--SDVS delete (task loop):       begin

      -- Waiting for a message to be sent from CP and
      -- built to process
```

```
--SDVS delete (writestring, writeln):      WRITESTRING("apPM pends");  WRITELN;


--SDVS comment: The next statement is an extended rendezvous with the task
--SDVS comment: MANAGE_MSG_RETRIEVAL at the entry  RET_NEXT_MSG. The rendezvous
--SDVS comment: can take place only if App_Msg_Counter > 0. Hence, we have
--SDVS comment: replaced the rendezvous with an if statement that executes
--SDVS comment: the code following the rendezvous only if App_Msg_Counter > 0.

--SDVS delete (task):      MANAGE_MSG_RETRIEVAL.RET_NEXT_MSG(New_App_Msg);
  if App_Msg_Counter > 0 then --SDVS replace
      MANAGE_MSG_RETRIEVAL(New_App_Msg); --SDVS replace

--SDVS delete (writestring, writeln):      WRITESTRING("apPMp");  WRITELN;

      -- The following code will be executed only if
      -- MANAGE_MSG_RETRIEVAL.RET_NEXT_MSG has been accepted
      -- (or rendezvoused).
      --
      -- Typical Message Format
      --        +--------------------+--------------------+
      --   (1)  |                    |      op-code       |
      --        +--------------------+--------------------+
      --   (2)  |               msg word                  |
      --        +--------------------+--------------------+
      --                             .
      --                             .
      --                             .
      --        |                    |                    |
      --        +--------------------+--------------------+
      --   (n)  |             msg checksum                |
      --        +--------------------+--------------------+
      --                             .
      --                             .
      --                             .
      --        |                    |                    |
      --        +--------------------+--------------------+
      --   (80)|          size of msg words = n           |
      --        +--------------------+--------------------+
      --

      New_Cmd_Op_Code := CONV_INTEGER_TO_BYTE(NEW_APP_MSG(1));

      Curr_Mode_State := MODE_STATE.RETRIEVE_CUR_STATE;

      case New_Cmd_Op_Code is


        when Data_Struc_Load =>

        --  Data Struc Load Message Format
        --      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 1    | 0   0   0   0   0   0   0   0 |         opcode = 1            |
        --      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 2    | 0   0   0   0   0   0 | E | R |       data structure id       |
        --      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 3  |                         data structure word                    |
        --      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        --    |                         data structure word                    |
        --      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        --                                       .
        --                                       .
        --                                       .
        --      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        --    |                              checksum                          |
        --      +===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+
        --    | /   /   /   /   /   /   /   /   /   /   /   /   /   /   /   / |
        --      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        --                                       .
        --                                       .
        --                                       .
        --      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 80 |                          num of msg words                      |
        --      +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- E : 1 - store in EEPROM
        -- R : 1 - store in RAM

--SDVS comment: The call below to MEMORY_MANAGER.CONV_TYPE is used to assign
--SDVS comment: the value of New_App_Msg(2) (an integer) to  Temp_Word_1
--SDVS comment: (a word). The body of CONV_TYPE is written in 1750A. We replace
--SDVS comment: this call by a type conversion.
--SDVS delete (1750A):         MEMORY_MANAGER.CONV_TYPE
--SDVS delete (1750A):                (New_App_Msg(2)'address,    -- Source
--SDVS delete (1750A):                 Temp_Word_1'address,            -- Target
--SDVS delete (1750A):                 1);                            -- Words

          Temp_Word_1 := GLOBAL_TYPES.WORD(New_App_Msg(2)); --SDVS replace

--SDVS delete (hex):           Temp_Word    := AND_WORD(16#FF00#, Temp_Word_1);
          Temp_Word    := AND_WORD(65280, Temp_Word_1); --SDVS replace

--SDVS delete (hex):           Temp_Word_2  := AND_WORD(16#00FF#, Temp_Word_1);
          Temp_Word_2  := AND_WORD(255, Temp_Word_1);

          Temp_Word_1 := SHIFT_LOGICAL_WORD(Temp_Word, -8);
          Storage_Selection := CONV_WORD_TO_INTEGER(Temp_Word_1);

          Data_Struc_ID     := CONV_WORD_TO_INTEGER(Temp_Word_2);
          Data_Struc_Length := TRKING_DATA_STRUC.Words_For_Data_Struc(
                                                          Data_Struc_ID);
          if Data_Struc_Length /= 0 then

            if (Curr_Mode_State /= Init_Memory_Load) and
               (Curr_Mode_State /= Init_Memory_Dump) then

              if (Storage_Selection = Update_EEPROM) or
                 (Storage_Selection = Update_Both) then

                MODE_STATE.CHG_STATE(Init_EEPROM_Write, Status_Of_Request);
                if Status_Of_Request = State_Granted then

                  -- Strip off New_App_Msg(1) which contains opcode,
                  -- the contents of data structure (including checksum)is
                  -- stored in New_App_Msg(2) thru (Data_Struc_Length+1).
                  --New_App_Msg(1..Data_Struc_Length) :=
                  --                New_App_Msg(2..Data_Struc_Length+1);
--SDVS delete (array slice):       EEPROM_DATA.STORE_DATA_STRUC_IN_EEPROM
                  --          (New_App_Msg(1..Data_Struc_Length),
--SDVS delete (array slice):         (New_App_Msg(2..Data_Struc_Length+1),
--SDVS delete (array slice):          Data_Struc_ID,
--SDVS delete (array slice):          Data_Struc_Length);
                  EEPROM_DATA.STORE_DATA_STRUC_IN_EEPROM
                            (New_App_Msg,
                             Data_Struc_ID,
                             Data_Struc_Length);
                  MODE_STATE.CHG_STATE(Curr_Mode_State, Status_Of_Request);
                  if Status_Of_Request /= State_Granted then
                    -- State_Not_Restored_After_EEPROM_Wr = 43.
                    TM_DATA.MANAGE.REPORT_SYSTEM_ERROR (43);
                  end if;

                else  -- Init_EEPROM_Write state not granted.
                  -- Data_Struc_Lost_EEPROM_State_Denied = 44
                  TM_DATA.MANAGE.REPORT_SYSTEM_ERROR (44);
                end if;

              end if;

              if    (Storage_Selection = Update_RAM) or
                    (Storage_Selection = Update_Both) then
                -- Mask storage selection field before passing to
                -- store in RAM.
--SDVS delete   New_App_Msg(2) := Data_Struc_ID;

                -- Strip off checksum word.
                --New_App_Msg(1..Data_Struc_Length-1) :=
                --                    New_App_Msg(2..Data_Struc_Length);
                --TRKING_PARAMS.MANAGE.STORE_TRKING_STRUC(
                --                         New_App_Msg(1..Data_Struc_Length-1));
--SDVS delete  TRKING_PARAMS.MANAGE.STORE_TRKING_STRUC(
--SDVS delete                          New_App_Msg(2..Data_Struc_Length));
                null; --SDVS replace

              end if;

              if (Storage_Selection > Update_Both) and
                  (Storage_Selection < Update_RAM) then
                 -- Data_Struc_Storage_Invalid = 45
                TM_DATA.MANAGE.REPORT_SYSTEM_ERROR (45);
              end if;

            else  -- Current mode is either init memory dump or memory load.
              -- Data_Struc_Lost_Invalid_State = 46
              TM_DATA.MANAGE.REPORT_SYSTEM_ERROR (46);
              end if;

            end if;  -- Data struc length is not 0.  Length 0 implies unused
                     -- data struc ID.


          when Sat_Subsys_Config =>
            null; --SDVS add
          --  Satellite Subsystem Configuration  Message Format
          --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
          -- 1   | 0   0   0   0   0   0   0   0 |         opcode = 22           |
          --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
          -- 2   | /   /   /   /   /   /   /   / | reserved | A | B | C | D | E |
          --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
          -- 3   | /   /   /   /   /   /   /   / | 0   0   0   0   0   0 | F | G |
          --     +===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+
          --     | /   /   /   /   /   /   /   /   /   /   /   /   /   /   /   / |
          --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
          --                                     .
          --                                     .
          --                                     .
          --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
          -- 80  |                     num of msg words = 3                      |
          --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
          -- A : 0 - AP 1
          --     1 - AP 2
          -- B : 0 - 25  MBPS
          --     1 - 5 MBPS
          -- C : 0 - CP 1
          --     1 - CP 2
          -- D : 0 - Upload 1
          --     1 - Upload 2
          -- E : 0 - DHS 1
          --     1 - DHS 2
          -- F : 0 - TP 1
          --     1 - TP 2
          -- G : 0 - Do not save configuration word in EEPROM
          --     1 - Save configuration word in EEPROM

--SDVS delete:          if Curr_Mode_State = Init_Setup or
--SDVS delete:             Curr_Mode_State = Tracking_Setup or
--SDVS delete:             Curr_Mode_State = Tracking then


--SDVS delete:            MEMORY_MANAGER.CONV_TYPE
--SDVS delete:                    (New_App_Msg(2)'address,    -- Source
--SDVS delete:                     Temp_Word'address,                -- Target
--SDVS delete:                     1);                        -- Words
--SDVS delete:            Subsys_Config_Bits := SHIFT_LOGICAL_WORD(Temp_Word, 8);

--SDVS delete:            MEMORY_MANAGER.CONV_TYPE
--SDVS delete:                    (New_App_Msg(3)'address,    -- Source
--SDVS delete:                     Temp_Word'address,         -- Target
--SDVS delete:                     1);                        -- Words
--SDVS delete:            TP_Num_And_Config_Save_Bits := AND_WORD(Temp_Word, 16#0003#);
```

```
--SDVS delete:            OUTPUT_WORD(Subsys_Config_Bits, PO, 16#0028#);

--SDVS delete:            Saved_Sat_Subsys_Config
--SDVS delete:                := OR_WORD(Subsys_Config_Bits, TP_Num_And_Config_Save_Bits);

            -- Save the configuration word in EEPROM if flagged to
            -- do so and also provide it to TM for reporting.

--SDVS delete:            if (New_App_Msg(3) = 1) or
--SDVS delete:               (New_App_Msg(3) = 3) then
--SDVS delete:              EEPROM_DATA.STORE_SYS_CONFIG(Saved_Sat_Subsys_Config);
--SDVS delete:            end if;
--SDVS delete:
            -- Set prime science data rate.

--SDVS delete:            Temp_Word := AND_WORD(Saved_Sat_Subsys_Config, 16#0800#);
--SDVS delete:            if Temp_Word = 16#0800# then
--SDVS delete:              PRIME_SCIENCE_DATA.SET_PRIME_SCI_DATA_RATE(5);
--SDVS delete:            else
--SDVS delete:              PRIME_SCIENCE_DATA.SET_PRIME_SCI_DATA_RATE(25);
--SDVS delete:            end if;

--SDVS delete:          else  -- current state inhibit satellite subsystem config change.
            -- Subsys_Config_Chg_Not_Allowed = 5
--SDVS delete:            TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(5);
--SDVS delete:          end if;


        when Soft_Reset =>
            null; --SDVS add
        --  Soft Reset Message Format
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 1  | 0   0   0   0   0   0   0   0 |        opcode = 32            |
        --    +===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+
        -- 2  | /   /   /   /   /   /   /   /   /   /   /   /   /   /   /   / |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        --                                    .
        --                                    .
        --                                    .
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 80 |                     num of msg words = 1                      |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

          -- Soft_Reset message is allowed in any mode/state except
          -- in Init_EEPROM_Write state which cannot happen when
          -- processing this message since enter and exit Init_EEPROM_
          -- Write state are required while storing an message in EEPROM.

--SDVS delete:          MODE_STATE.PERFORM_SOFT_RESET;


        when Prep_For_Long_Load =>
        --  Prepare For Long Load Message Format
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 1  | 0   0   0   0   0   0   0   0 |       opcode = 11             |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 2  | /   /   /   /   /   /   /   / | 0   0   0   0   0   0 |   A   |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 3  | /   /   /   /   /   /   /   / |        num of blocks          |
        --    +===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+
        --    | /   /   /   /   /   /   /   /   /   /   /   /   /   /   /   / |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        --                                    .
        --                                    .
        --                                    .
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 80 |                      num of msg words = 3                     |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- A: 1 - command system 1
        --    2 - command system 2
          null; --SDVS add
--SDVS delete:          MODE_STATE.CHG_STATE(Init_Memory_Load, Status_Of_Request);

--SDVS delete:          if Status_Of_Request = State_Granted then

--SDVS delete:            Memory_Upload_Select := AND_I(New_App_Msg(2), 16#0003#);

--SDVS delete:            if Memory_Upload_Select = 1 then

             -- configure for memory upload 1
--SDVS delete:              Saved_Sat_Subsys_Config := AND_WORD
--SDVS delete:                                         (Saved_Sat_Subsys_Config, 16#FD03#);
--SDVS delete:            elsif Memory_Upload_Select = 2 then

             -- configure for memory upload 2
--SDVS delete:              Saved_Sat_Subsys_Config := OR_WORD
--SDVS delete:                                         (Saved_Sat_Subsys_Config, 16#0200#);
--SDVS delete:            end if;

--SDVS delete:            OUTPUT_WORD(Saved_Sat_Subsys_Config, P0, 16#0028#);

           -- Initiate long memory load process.
--SDVS delete:            ARRAY_OF_BLOCKS.INIT_LONG_LOAD(New_App_Msg(3));
--SDVS delete:            ARRAY_OF_BLOCKS.PROCESS_MSGS_IN_BLOCKS.PREPARE_FOR_TIMEOUT;

--SDVS delete:          else  -- state not granted
           -- Init_Mem_Load_Sta_Denied = 2
--SDVS delete:            TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(2);
--SDVS delete:          end if;  -- state granted or denied


        when Mem_Dump_X_Frames =>
           null; --SDVS add
        --  Memory Dump X Frames Message Format
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 1  | 0   0   0   0   0   0   0   0 |       opcode = 13             |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 2  | 0   0   0   0   0   0   0   0 |  num of dump data words = k   |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 3  | 0   0   0   0   0   0   0   0 | 0   0   0   0 | high addr bits|
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 4  |                          low addr bits                        |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 5  |                           checksum                            |
        --    +===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+
        --    | /   /   /   /   /   /   /   /   /   /   /   /   /   /   /   / |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        --                                .
        --                                .
        --                                .
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 80 |                      num of msg words = 5                     |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

--SDVS delete:          MODE_STATE.CHG_STATE(Init_Memory_Dump, Status_Of_Request);

--SDVS delete:          if Status_Of_Request = State_Granted then

--SDVS delete:            MEMORY_MANAGER.CONV_TYPE(New_App_Msg(3)'address,
--SDVS delete:                                     Dump_Address'address,
--SDVS delete:                                     2);

--SDVS delete:            Dump_Params := (Kind_Of_Dump  => Spec_Addr,
--SDVS delete:                            Num_Frames    => NEW_APP_MSG(2),
--SDVS delete:                            Start_Address => Dump_Address);

--SDVS delete:            TM.PROCESS_CMD.SET_MEM_DUMP_PARAMS(Dump_Params);

--SDVS delete:          else  -- Status_Of_Request = State_Denied
            -- Init_Mem_Dump_Sta_Denied = 3
--SDVS delete:            TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(3);

--SDVS delete:          end if;


        when Shutdown_Imminent =>
           null; --SDVS add
        --  Shutdown Imminent  Message Format
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 1  | 0   0   0   0   0   0   0   0 |         opcode = 19           |
        --    +===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+
        -- 2  | /   /   /   /   /   /   /   /   /   /   /   /   /   /   /   / |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        --                                .
        --                                .
        --                                .
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 80 |                      num of msg words = 1                     |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

--SDVS delete:          POINTING_INFO_OUT.SET_SHUTDOWN_IMMINENT;


        when Chg_Monitor_Loc =>
```

```
                   null; --SDVS add
                --  Change Monitor Location Message Format
                --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                -- 1   | 0   0   0   0   0   0   0   0 |         opcode = 28           |
                --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                -- 2   | 0   0   0   0   0   0   0   0   0   0   0   0 | high addr bits|
                --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                -- 3   |                          low addr bits                        |
                --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                -- 4   | 0   0   0   0   0   0   0   0   0   0   0   0 | high addr bits|
                --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                -- 5   |                          low addr bits                        |
                --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                -- 6   | 0   0   0   0   0   0   0   0   0   0   0   0 | high addr bits|
                --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                -- 7   |                          low addr bits                        |
                --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                -- 8   | 0   0   0   0   0   0   0   0   0   0   0   0 | high addr bits|
                --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                -- 9   |                          low addr bits                        |
                --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                -- 10  | 0   0   0   0   0   0   0   0   0   0   0   0 | high addr bits|
                --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                -- 11  |                          low addr bits                        |
                --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                -- 12  | 0   0   0   0   0   0   0   0   0   0   0   0 | high addr bits|
                --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                -- 13  |                          low addr bits                        |
                --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                -- 14  |                            checksum                           |
                --     +===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+
                --     | /   /   /   /   /   /   /   /   /   /   /   /   /   /   /   / |
                --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                --                                     .
                --                                     .
                --                                     .
                --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                -- 80  |                     num of msg words = 14                     |
                --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

--SDVS delete:          if Curr_Mode_State = Init_Setup or
--SDVS delete:             Curr_Mode_State = Tracking_Setup or
--SDVS delete:             Curr_Mode_State = Tracking then

              -- Convert integers to an array of addresses
--SDVS delete:            MEMORY_MANAGER.CONV_TYPE
--SDVS delete:                     (New_App_Msg(2)'address,
--SDVS delete:                      Mem_Locations_To_Monitor(1)'address,
--SDVS delete:                      12);

--SDVS delete:            TM.PROCESS_CMD.UPDATE_VAR_MON_ADDR(Mem_Locations_To_Monitor);

--SDVS delete:          else
              -- Chg_Monitor_Loc_Denied = 6
--SDVS delete:            TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(6);
```

```
--SDVS delete:          end if;


        when No_Op_Cmd =>
           null; --SDVS add
        --  No-Operation  Message Format
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 1  | 0   0   0   0   0   0   0   0 |        opcode = 25            |
        --     +===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+
        -- 2  | /   /   /   /   /   /   /   /   /   /   /   /   /   /   /   / |
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        --                                     .
        --                                     .
        --                                     .
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 80 |                      num of msg words = 1                     |
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

--SDVS delete:                  if Curr_Mode_State = Init_Setup or
--SDVS delete:             Curr_Mode_State = Init_Memory_Load or
--SDVS delete:             Curr_Mode_State = Tracking_Setup or
--SDVS delete:             Curr_Mode_State = Tracking then

--SDVS delete:            TM.PROCESS_CMD.CHK_NO_OP_CMD;

--SDVS delete:          else
            -- No_Op_Cmd_Denied = 7
--SDVS delete:            TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(7);

--SDVS delete:          end if;


        when RAM_Mem_Load =>
           null; --SDVS add
        --  RAM Memory Load Message Format
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 1  | 0   0   0   0   0   0   0   0 |        opcode = 2             |
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 2  | 0   0   0   0   0   0   0   0 |   num of load data words = k  |
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 3  | 0   0   0   0   0   0   0   0 | 0   0   0   0 | high addr bits|
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 4  |                          low addr bits                        |
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 5  |                          load data word 1                     |
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 6  |                          load data word 2                     |
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        --                                     .
        --                                     .
        --                                     .
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- K+4|                          load data word k                     |
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- K+5|                                 checksum                              |
        --    +===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+
        --    | /   /   /   /   /   /   /   /   /   /   /   /   /   /   /   / |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        --                                    .
        --                                    .
        --                                    .
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 80 |                        num of msg words = K+5                 |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

--SDVS delete:          if Curr_Mode_State = Init_Setup or
--SDVS delete:             Curr_Mode_State = Tracking_Setup or
--SDVS delete:             Curr_Mode_State = Tracking then

--SDVS delete:            MEMORY_MANAGER.CONV_TYPE(New_App_Msg(3)'address,
--SDVS delete:                                     Addr_Of_RAM_Load'address,
--SDVS delete:                                     2);

--SDVS delete:            Last_Addr_Load := Addr_Of_RAM_Load +
--SDVS delete:                              LONG_INTEGER(New_App_Msg(2)) - 1;

              -- REMINDER: need logic to verfy that the Addr_Of_RAM_Load
              -- is not violating the restricted RAM (air?) space.

--SDVS delete:            if Last_Addr_Load   <= 16#7FFFF# and
--SDVS delete:               Addr_Of_RAM_Load >= 0 then

              -- pass destination addr, data and number of words.
--SDVS delete:              MEMORY_MANAGER.WRITE_TO_RAM
--SDVS delete:                             (Addr_Of_RAM_Load,
--SDVS delete:                              NEW_APP_MSG(5)'ADDRESS,
--SDVS delete:                              New_App_Msg(2));

--SDVS delete:            else -- load addr is beyond RAM addr
              -- RAM_Load_Addr_Invalid = 10
--SDVS delete:              TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(10);
--SDVS delete:            end if;  -- load addr is within or beyond RAM addr

--SDVS delete:          else
            -- not allowed during memory dump or long load
            -- RAM_Mem_Load_Denied = 9
--SDVS delete:            TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(9);
--SDVS delete:          end if;


        when EEPROM_Mem_Load =>
           null; --SDVS add
        --  EEPROM Memory Load Message Format
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 1  | 0   0   0   0   0   0   0   0 |        opcode = 4             |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 2  | 0   0   0   0   0   0   0   0 |  num of load data words = k   |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 3  | 0   0   0   0   0   0   0   0 | 0   0   0   0 | high addr bits|
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 4  |                         low addr bits                         |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 5  |                         load data word 1                      |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 6  |                         load data word 2                      |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        --                                    .
        --                                    .
        --                                    .
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- K+4|                         load data word k                      |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- K+5|                           checksum                            |
        --    +===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+
        --    | /   /   /   /   /   /   /   /   /   /   /   /   /   /   /   / |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        --                                    .
        --                                    .
        --                                    .
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 80 |                     num of msg words = k+5                    |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

--SDVS delete:          MODE_STATE.CHG_STATE(Init_EEPROM_Write, Status_Of_Request);

--SDVS delete:          if Status_Of_Request = State_Granted then


            -- Strip off New_App_Msg(1), opcode word
--SDVS delete:            MEMORY_MANAGER.CONV_TYPE(New_App_Msg(2)'address,
--SDVS delete:                                     Load_Msg(1)'address,
--SDVS delete:                                     CMDS_TYPES.App_Msg_Max-1);

--SDVS delete:            Load_Msg_Is_Data_Struc := False;
--SDVS delete:            EEPROM_DATA.WRITE_LOAD_DATA_TO_EEPROM
--SDVS delete:                       (Load_Msg,
--SDVS delete:                        Load_Msg_Is_Data_Struc,
--SDVS delete:                        Stored);

--SDVS delete:            MODE_STATE.CHG_STATE(Curr_Mode_State, Status_Of_Request);
--SDVS delete:            if Status_Of_Request /= State_Granted then
              -- State_Not_Restored_After_EEPROM_Wr = 43
--SDVS delete:              TM_DATA.MANAGE.REPORT_SYSTEM_ERROR (43);
--SDVS delete:            end if;

--SDVS delete:          else  -- EEPROM write state denied
            -- Init_EEPROM_W_Sta_Denied = 8
--SDVS delete:            TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(8);
--SDVS delete:          end if;  -- EEPROM Write state granted or denied


        when Set_RAM_Def_Tab_Entry =>

          -- COMING SOON, may not come forever
          null;

        when Clr_Timeout_Telltale =>
          null; --SDVS add
        --  Clear Timeout Telltale  Message Format
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 1   | 0   0   0   0   0   0   0   0 |        opcode = 37            |
        --     +===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+
        -- 2   | /   /   /   /   /   /   /   /   /   /   /   /   /   /   /   / |
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        --                                     .
        --                                     .
        --                                     .
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 80  |                       num of msg words = 1                    |
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

--SDVS delete:          OUTPUT_WORD(16#0000#, P0, 16#0018#);

        when UT_Control =>
          null; --SDVS add
        --  UT Control Message Format
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 1   | 0   0   0   0   0   0   0   0 |        opcode = 38            |
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 2   | /   /   /   /   /   /   /   / | 0   0   0   0   0   0   0   0 |
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 3   | /   /   /   /   /   /   /   / | 0   0   0   0   0   0   0 | I |
        --     +===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+
        -- 4   | /   /   /   /   /   /   /   /   /   /   /   /   /   /   /   / |
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        --                                     .
        --                                     .
        --                                     .
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 80  |                       num of msg words = 3                    |
        --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- I : 1 - UT interal control
        --     0 - UT normal control

--SDVS delete:          if New_App_Msg(3) = UT_Internal then
--SDVS delete:            OUTPUT_WORD(16#0000#, P0, 16#0260#);
--SDVS delete:          else
--SDVS delete:            OUTPUT_WORD(16#0000#, P0, 16#0250#);
--SDVS delete:          end if;

        when others =>

          -- Added to aid debugging.
          -- Invalid_Opcode_While_Processing = 102
          TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(102);
```

```
      end case;
    end if; --SDVS add -- if App_Msg_Counter > 0
--SDVS delete (exception):      exit;

--SDVS delete (exception):      exception

--SDVS delete (exception):        when tasking_error =>

--SDVS delete (exception):          writestring("appmte 1");
--SDVS delete (exception):          writeln;

--SDVS delete (exception):        when others =>

--SDVS delete (exception):          writestring("appme 2");
--SDVS delete (exception):          writeln;

--SDVS delete:      end; -- for begin

--SDVS delete (task loop):      end loop;

--SDVS delete (task loop):    end loop;

--SDVS delete (exception):    exception

--SDVS delete (exception):      when tasking_error =>

--SDVS delete (exception):        writestring("appmteol 3");
--SDVS delete (exception):        writeln;

--SDVS delete (exception):      when others =>

--SDVS delete (exception):        writestring("appmeol 4");
--SDVS delete (exception):        writeln;

end PROCESS_MSG;


------------------------------------------------------------------------------
--       TASK MANAGE_MSG_RETRIEVAL
------------------------------------------------------------------------------
-- PURPOSE :  This task returns the next application message
--            to be processed by the APP_MSGS.PROCESS_MSG.
--
--            APP_MSGS.PROCESS_MSG initiates a rendezvous with MANAGE_
--            MSG_RETRIEVAL at the entry point RET_NEXT_MSG.
--            APP_MSGS.BUILD initiates a rendezvous with MANAGE_MSG-
--            RETRIEVAL at the entry point NOTIFY_MSG_STORED when the
--            message queue goes from empty to a message stored.
--
-- NOTES:
--
-- CHANGE HISTORY:
--   10/16/91    B. Na      Initial Version
--   mm/dd/yy    B. Na      Build 2 Changes
```

```
--
--------------------------------------------------------------------------------
--SDVS delete (task):  task body MANAGE_MSG_RETRIEVAL_TYPE is
procedure MANAGE_MSG_RETRIEVAL(App_Msg_Out :
                                out CMDS_TYPES.APP_MSG_OUT_TYPE) is --SDVS replace

    Num_Words_Curr_App_Msg : INTEGER;

  begin

--SDVS delete (task loop):    loop

--SDVS delete (task loop):      loop -- for tasking_error
--SDVS delete (task loop):      begin

--SDVS delete (task):      select

--SDVS delete (task):        when App_Msg_Counter > 0 =>

--SDVS delete (task):          accept RET_NEXT_MSG(App_Msg_Out : out
--SDVS cont:                                    CMDS_TYPES.APP_MSG_OUT_TYPE) do

--SDVS delete (2-dimensional array):           Num_Words_Curr_App_Msg :=
--SDVS cont: CONV_WORD_TO_INTEGER (App_Msg_Q(Q_Head, CMDS_TYPES.App_Msg_Max));
      Num_Words_Curr_App_Msg :=
       CONV_WORD_TO_INTEGER (App_Msg_Q(Q_Head)(CMDS_TYPES.App_Msg_Max)); --SDVS replace



--SDVS delete (1750A):            MEMORY_MANAGER.CONV_TYPE
--SDVS delete (1750A):                      (App_Msg_Q(Q_Head, 1)'address,
--SDVS delete (1750A):                       App_Msg_Out(1)'address,
--SDVS delete (1750A):                       Num_Words_Curr_App_Msg);
            for index in 1 .. Num_Words_Curr_App_Msg loop --SDVS replace
             App_Msg_Out(index) := INTEGER(App_Msg_Q(Q_Head)(index)); --SDVS replace
            end loop; --SDVS replace
            -- save the number of words for application message at the
            -- bottom.



            App_Msg_Out(CMDS_TYPES.App_Msg_Max) := Num_Words_Curr_App_Msg;

            Q_Head := (Q_Head mod App_Msg_Q_Size) + 1;

           -- XIO RA,DSBL
--SDVS delete (artclient):            ARTCLIENT.ENTER_CRITICAL_SECTION;

            App_Msg_Counter := App_Msg_Counter - 1;

            -- XIO RA,ENBL enables int.
--SDVS delete (artclient):            ARTCLIENT.LEAVE_CRITICAL_SECTION;

--SDVS delete (task):          end RET_NEXT_MSG;
```

```
--SDVS delete (task):      or

--SDVS delete (task):        accept NOTIFY_MSG_STORED;

--SDVS delete (task);      end select;

--SDVS delete (task):      exit;

--SDVS delete (exception):      exception

--SDVS delete (exception):        when tasking_error =>

--SDVS delete (exception):          writestring("apmrte 1");
--SDVS delete (exception):          writeln;

--SDVS delete (exception):        when others =>

--SDVS delete (exception):          writestring("apmre 2");
--SDVS delete (exception):          writeln;

--SDVS delete:      end; -- for begin

--SDVS delete (task loop):      end loop;

--SDVS delete (task loop):    end loop;

--SDVS delete (exception):    null;

--SDVS delete (exception):   exception

--SDVS delete (exception):      when tasking_error =>

--SDVS delete (exception):        writestring("apmrteol 3");
--SDVS delete (exception):        writeln;

--SDVS delete (exception):      when others =>

--SDVS delete (exception):        writestring("apmreol 4");
--SDVS delete (exception):        writeln;

  end MANAGE_MSG_RETRIEVAL;




--SDVS comment: The function and two procedures that follow are never
--SDVS comment: called in this target code. They are
--SDVS comment: called by TM.COLLECT for telemetry reports on the system.
--SDVS comment: We have therefore deleted them.


----------------------------------------------------------------------------
```

```
--      FUNCTION NEW_TM_DATA
---------------------------------------------------------------------------
-- PURPOSE: This function returns TRUE, if new telemetry data is
--          available to be collected.  Otherwise, FALSE is returned.
--
-- NOTES:
--
-- CHANGE HISTORY:
--   10/16/91    B. Na      Initial Version
--
---------------------------------------------------------------------------
--SDVS delete:  function NEW_TM_DATA return BOOLEAN is

--SDVS delete:  begin

--SDVS delete:    if Read_Ptr = Write_Ptr then
--SDVS delete:      return FALSE;
--SDVS delete:            else
--SDVS delete:      return TRUE;
--SDVS delete:    end if;

--SDVS delete:  end NEW_TM_DATA;




---------------------------------------------------------------------------
--      PROCEDURE RET_LATEST_CMDS
---------------------------------------------------------------------------
-- PURPOSE:  To provide the latest 40 serial digital commands
--           received from CP to TM.COLLECT for reporting in the next
--           frame of the housekeeping telemetry.
--
-- NOTES:
--
-- CHANGE HISTORY:
--   11/11/91    B. Na      Initial Version
--
---------------------------------------------------------------------------
--SDVS delete:  procedure RET_LATEST_CMDS (Latest_Cmd_TM   : out LATEST_CMD_TM_TYPE) is

--SDVS delete:    Buf_Index      : INTEGER;
--SDVS delete:    Que_Index      : INTEGER;
--SDVS delete:    Temp_Read_Ptr  : INTEGER;
--SDVS delete:    Num_New_Cmds   : INTEGER;

--SDVS delete:  begin

--SDVS delete:    Temp_Read_Ptr := Read_Ptr;

    -- Determine the number of the latest cmds stored in queue
    -- and not reported in TM previously

--SDVS delete:    Num_New_Cmds := 0;
```

```
--SDVS delete:    while ((Temp_Read_Ptr /= Write_Ptr) and
--SDVS delete:           (Num_New_Cmds <= Latest_Cmds_Q_Size)) loop

--SDVS delete:      Temp_Read_Ptr := (Temp_Read_Ptr mod Latest_Cmds_Q_Size) + 1;
--SDVS delete:      Num_New_Cmds := Num_New_Cmds + 1;

--SDVS delete:    end loop;

--SDVS delete:    Latest_Cmd_TM.Fresh_Cmd_Count := Num_New_Cmds;



--SDVS delete:    if (Temp_Read_Ptr /= 0) and (Num_new_cmds <= Max_Num_Latest_Cmds) then

--SDVS delete:      Que_Index := Temp_Read_Ptr;
--SDVS delete:      for Buf_Index in 1..Max_Num_Latest_Cmds loop

--SDVS delete:        Latest_Cmd_TM.Cmd_Buf(Buf_Index)    := Latest_Cmds_Q(Que_Index);
--SDVS delete:        Latest_Cmd_TM.Status_Buf(Buf_Index) := Latest_Status_Q(Que_Index);

--SDVS delete:        Que_Index := Que_Index - 1;
--SDVS delete:        if Que_Index <= 0 then
--SDVS delete:          Que_Index := Latest_Cmds_Q_Size;
--SDVS delete:        end if;

--SDVS delete:      end loop;

--SDVS delete:      Read_Ptr := Temp_Read_Ptr;

--SDVS delete:    elsif (Temp_Read_Ptr /= 0) then  -- if number of latest cmds is
                                       -- more than 40
--SDVS delete:      Temp_Read_Ptr := Read_Ptr;
--SDVS delete:      for Buf_Index in reverse 1..Max_Num_Latest_Cmds loop

--SDVS delete:        Temp_Read_Ptr := (Temp_Read_Ptr mod Latest_Cmds_Q_Size) + 1;
--SDVS delete:        Latest_Cmd_TM.Cmd_Buf(Buf_Index)    := Latest_Cmds_Q(Temp_Read_Ptr);
--SDVS delete:        Latest_Cmd_TM.Status_Buf(Buf_Index) := Latest_Status_Q(Temp_Read_Ptr);
--SDVS delete:
--SDVS delete:      end loop;

--SDVS delete:      Read_Ptr := Temp_Read_Ptr;

--SDVS delete:    else  -- if Read_Ptr = 0, no cmd is received since power up

      -- Set to something to avoid exception.
--SDVS delete:      for Buf_Index in 1..Max_Num_Latest_Cmds loop
--SDVS delete:        Latest_Cmd_TM.Cmd_Buf(Buf_Index)    := Latest_Cmds_Q(Buf_Index);
--SDVS delete:        Latest_Cmd_TM.Status_Buf(Buf_Index) := Latest_Status_Q(Buf_Index);
--SDVS delete:      end loop;

--SDVS delete:    end if;


    -- Note:  Cmd_Total is not always consistent/correlated with
    --        Rejected_Cmd_Count because increase Cmd_Total by
    --        Cmd_Count and later individually process cmds to see
    --        if should increment Rejected_Cmd_Count.

--SDVS delete:    Latest_Cmd_TM.Cmd_Total                := APP_MSGS.Cmd_Total;
--SDVS delete:    Latest_Cmd_TM.Rejected_Cmd_Count       := APP_MSGS.Rejected_Cmd_Count;
--SDVS delete:    Latest_Cmd_TM.IO_Status_Word           := APP_MSGS.IO_Status_Word;
--SDVS delete:    Latest_Cmd_TM.FIFO_Not_Empty_Count := APP_MSGS.FIFO_Not_Empty_Count;
--SDVS delete:    Latest_Cmd_TM.Saved_Sat_Subsys_Config := APP_MSGS.Saved_Sat_Subsys_Config;

--SDVS delete:  end RET_LATEST_CMDS;




----------------------------------------------------------------------------
--      PROCEDURE RET_INIT_CMD_LAST_APP_MSG
----------------------------------------------------------------------------
-- PURPOSE:  To provide the starting serial digital command
--           corresponding to the last application message to TM.COLLECT
--           for reporting in the next frame of the housekeeping telemetry.
--
-- NOTES:
--
-- CHANGE HISTORY:
--   11/11/91    B. Na      Initial Version
--
----------------------------------------------------------------------------
--SDVS delete:  procedure RET_INIT_CMD_LAST_APP_MSG
--SDVS delete:                (Last_App_Msg_TM : out LAST_APP_MSG_TM_TYPE) is

--SDVS delete:  begin

--SDVS delete:    Last_App_Msg_TM.Starting_Cmd          := APP_MSGS.Cmd_Last_App_Msg;
--SDVS delete:    Last_App_Msg_TM.Cmd_Total             := APP_MSGS.Cmd_Total;
--SDVS delete:    Last_App_Msg_TM.Rejected_Cmd_Count    := APP_MSGS.Rejected_Cmd_Count;
--SDVS delete:    Last_App_Msg_TM.FIFO_Not_Empty_Count := APP_MSGS.FIFO_Not_Empty_Count;
--SDVS delete:    Last_App_Msg_TM.IO_Status_Word        := APP_MSGS.IO_Status_Word;
--SDVS delete:    Last_App_Msg_TM.Saved_Sat_Subsys_Config := APP_MSGS.Saved_Sat_Subsys_Config;

--SDVS delete:  end RET_INIT_CMD_LAST_APP_MSG;


--SDVS comment: The function RET_MSG_LENGTH returns the length of the next block
--SDVS comment: of commands in the input based on the message identifier. We
--SDVS comment: added it for the scheduler.
function RET_MSG_LENGTH(Msg_Id: GLOBAL_TYPES.BYTE) return INTEGER is --SDVS add
        begin --SDVS add
          return Cmds_For_Data_Struc(CONV_BYTE_TO_INTEGER(Msg_Id)); --SDVS add
        end RET_MSG_LENGTH; --SDVS add



----------------------------------------------------------------------------
--          INITIATE_BUILD
```

```
---------------------------------------------------------------------------
-- PURPOSE :  This procedure initiates the application message
--            building when called by APP_MSGS.BUILD.  The rules for
--            building an app_msg to verify the checksum are as follows:
--
--              1) app_msg(1) is blank.  This is so that after the
--                  checksum is verified, the header for RAM_mem_Load,
--                  EEPROM_mem_load, and Mem_Dump_X_Frames can be
--                  easily modified to store the address in 32 bits.
--
--              2) app_msg(2) contains the initial data op-code for
--                 the application msg.
--
--              3) if more than one command is required for an application
--                 message then the data associated with the checksum is
--                 stored in app_msg(3)..app_msg(n).  App_msg(n) is a
--                 complete word.  There is not a spare byte at the end.
--                 App_msg(n+1) contains the checksum.
--
-- EXCEPTIONS:
--
-- NOTES:
--
-- CHANGE HISTORY:
--   10/16/91    B. Na      Initial Version
--   mm/dd/yy    B. Na      Build 2 Changes
--
---------------------------------------------------------------------------
procedure INITIATE_BUILD (Start_Op_Code : in GLOBAL_TYPES.BYTE;
                          Cmd_Buf_Index : in INTEGER) is

  -- Used to hold intermediate values for bitwise operations.
  Temp_word                 : GLOBAL_TYPES.WORD;
  High_Byte_Word            : GLOBAL_TYPES.WORD;
  Low_Byte_Word             : GLOBAL_TYPES.WORD;

  Data_Struc_ID_Found       : BOOLEAN;
  Start_Build               : BOOLEAN;

  Struc_ID_Index            : INTEGER;

  -- Used to hold intermediate values during the calcualtions of the
  -- number of commands and number of words for for the memory load
  -- messages.
  X                         : INTEGER;
  Y                         : INTEGER;


begin

  Start_Build := False;

  case Start_Op_Code is

    when Data_Struc_Load =>
    -- See message format for Group 1 commands below.

    Data_Struc_ID_Found := False;
    Struc_ID_Index :=
        CONV_BYTE_TO_INTEGER(Cmd_Buf(Cmd_Buf_Index).Third_Byte);

    if (Struc_ID_Index <= TRKING_DATA_STRUC.Num_Data_Struc) and
       (Struc_ID_Index > 0) then

      Cont_Cmd := Data_Load_Cmd;
      Num_Cmds_For_App_Msg  := Cmds_For_Data_Struc(Struc_ID_Index);
      Num_Words_For_App_Msg := TRKING_DATA_STRUC.Words_For_Data_Struc(
                                                     Struc_ID_Index) + 1;

      -- The look-up table, Cmds_For_Data_Struc, holds a value 0
      -- 0 if the corresponding data structure does not exist.
      if (Num_Cmds_For_App_Msg /= 0) then
        Data_Struc_ID_Found := True;
        Start_Build := True;
      end if;
    end if;

    if (Data_Struc_ID_Found = False) then

      Cmd_Status_Buf(Cmd_buf_index) := Invalid_Data_Struct_Msg_ID;
      -- Data_Struct_ID_Not_Found = 13
      TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(13);
      Rejected_Cmd_Count :=
               (Rejected_Cmd_Count + 1) mod Max_Rejected_Cmd_Count;
    end if;


  when RAM_Mem_Load | EEPROM_Mem_Load =>

    -- See message format for Group 1 commands below.


    -- X is the number of load data words specified in the command.
    X := CONV_BYTE_TO_INTEGER(Cmd_Buf(Cmd_buf_index).Second_Byte);

    -- Limit the number of load data words to 75 here to prevent
    -- a constraint error (App_Msg is defined for 80 words).
    if (X > 0) and (X <= 75) then

      Cont_Cmd := Data_Load_Cmd;

      -- New Method:
      --    Y := 2 + 2X / 3    where X = num of load words, > 0
      --    Num_Cmds_For_App_Msg := Y      if remainder =  0
      --    Num_Cmds_For_App_Msg := Y + 1 if remainder /= 0

      Y := 2 + ((2 * X )/ 3);
      if (X rem 3) = 0 then
        Num_Cmds_For_App_Msg := Y;
        else
          Num_Cmds_For_App_Msg := Y + 1;
        end if;

        Num_Words_For_App_Msg := X + 4;

        Start_Build := True;

      else  -- Invalid number of load data words.

        -- Invalid_Num_Of_Load_Data_Words = 103
        TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(103);

        Rejected_Cmd_Count :=
                 (Rejected_Cmd_Count + 1) mod Max_Rejected_Cmd_Count;
      end if;


    when Set_RAM_Def_Tab_Entry =>

      -- COMING SOON or maybe never
      null;


    when Prep_For_Long_Load | Sat_Subsys_Config | UT_Control =>

      if App_Msg_Counter < App_Msg_Q_Size then

        -- Message Format For Group 2 Commands
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 1  | 0   0   0   0   0   0   0   0 |   cmd byte 1 without parity   |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 2  | /   /   /   /   /   /   /   / |        cmd byte 2             |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 3  | /   /   /   /   /   /   /   / |        cmd byte 4             |
        --    +===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+
        --    | /   /   /   /   /   /   /   /   /   /   /   /   /   /   /   / |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        --                                    .
        --                                    .
        --                                    .
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        -- 80 |                      num of msg words = 3                     |
        --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
        --
        -- Note:  /  represents bits not initialized.
        --          cmd byte 3 is not used.

        -- Save the command associated with the last application message,
        -- so it can be reported to TM.
        Cmd_Last_App_Msg  := Cmd_Buf(Cmd_buf_index);

        Q_Tail := (Q_Tail mod App_Msg_Q_Size) + 1;

--SDVS delete (2-dimensional array): App_Msg_Q(Q_Tail, 1) :=
--SDVS delete                                CONV_BYTE_TO_WORD(Start_Op_Code);
        App_Msg_Q(Q_Tail)(1) := CONV_BYTE_TO_WORD(Start_Op_Code); --SDVS replace

--SDVS delete (2-dimensional array): App_Msg_Q(Q_Tail, 2) :=
--SDVS delete (2-dimensional array): CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_buf_index).Second_Byte);
        App_Msg_Q(Q_Tail)(2) := --SDVS replace
           CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_buf_index).Second_Byte); --SDVS replace


--SDVS delete (2-dimensional array): App_Msg_Q(Q_Tail, 3) :=
--SDVS delete (2-dimensional array): CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_buf_index).Fourth_Byte);
        App_Msg_Q(Q_Tail)(3) :=         --SDVS replace
           CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_buf_index).Fourth_Byte); --SDVS replace



--SDVS delete (2-dimensional array):  App_Msg_Q(Q_Tail, CMDS_TYPES.App_Msg_Max) := 3;
        App_Msg_Q(Q_Tail)(CMDS_TYPES.App_Msg_Max) := 3;         --SDVS replace



        -- XIO RA,DSBL disables int.
--SDVS delete (artclient):       Artclient.Enter_Critical_Section;

        -- *** REMINDER ***
        -- Check assembly to make sure App_Msg_Counter read from RAM
        -- (i.e. current value of App_Msg_Counter) not from register.

        App_Msg_Counter := App_Msg_Counter + 1;

        -- XIO RA,ENBL enables interrupt.
--SDVS delete (artclient):       Artclient.Leave_Critical_Section;

      else  -- queue is full
        -- App_Msg_Q_Overflowed = 17
        TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(17);
        Rejected_Cmd_Count :=
             (Rejected_Cmd_Count + 1) mod Max_Rejected_Cmd_Count;
        Cmd_Status_Buf(Cmd_buf_index)  := App_Msg_Dropped;

      end if;


    when Mem_Dump_X_Frames    =>

      -- See message format for Group 1 commands below.

      Cont_Cmd := Ext_Dump_X_Frames;

      Num_Cmds_For_App_Msg  := 2;
      Num_Words_For_App_Msg := 4;

      Start_Build := True;
      when No_op_Cmd | Clr_Timeout_Telltale =>

        if App_Msg_Counter < App_Msg_Q_Size then

          -- Message Format For Group 3 Commands
          --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
          -- 1   | 0   0   0   0   0   0   0   0 |   cmd byte 1 without parity   |
          --     +===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+
          --     | /   /   /   /   /   /   /   /   /   /   /   /   /   /   /   / |
          --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
          --                                     .
          --                                     .
          --                                     .
          --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
          -- 80  |                       num of msg words = 1                    |
          --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
          -- Note:  /  represents bits not initialized.
          --        cmd byte 2, 3 and 4 are not used.

          -- Save the command associated with the last application message,
          -- so it can be reported to TM.
          Cmd_Last_App_Msg := Cmd_Buf(Cmd_buf_index);

          Q_Tail := (Q_Tail mod App_Msg_Q_Size) + 1;

--SDVS delete (2-dimensional array):  App_Msg_Q(Q_Tail, 1) :=
--SDVS delete                        CONV_BYTE_TO_WORD(Start_Op_Code);
          App_Msg_Q(Q_Tail)(1) := CONV_BYTE_TO_WORD(Start_Op_Code); --SDVS replace


          -- store application message size for this type of commands
          -- at the end of application message array
--SDVS delete (2-dimensional array):  App_Msg_Q(Q_Tail, CMDS_TYPES.App_Msg_Max) := 1;
          App_Msg_Q(Q_Tail)(CMDS_TYPES.App_Msg_Max) := 1;        --SDVS replace

          -- XIO RA,DSBL disables int.
--SDVS delete (artclient):       Artclient.Enter_Critical_Section;

          -- *** REMINDER ***
          -- Check assembly to make sure App_Msg_Counter read from RAM
          -- (i.e. current value of App_Msg_Counter) not from register.

          App_Msg_Counter := App_Msg_Counter + 1;

--SDVS delete (artclient):       Artclient.Leave_Critical_Section;  -- XIO RA,ENBL

        else  -- queue is full
          -- App_Msg_Q_Overflowed = 17
          TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(17);
          Rejected_Cmd_Count :=
               (Rejected_Cmd_Count + 1) mod Max_Rejected_Cmd_Count;
          Cmd_Status_Buf(Cmd_buf_index) := App_Msg_Dropped;

        end if;
```

```
    when Chg_Monitor_Loc      =>

      -- See message format for Group 1 commands below.

      Cont_Cmd := Ext_Chg_Mon_Loc;

      Num_Cmds_For_App_Msg  := 9;
      Num_Words_For_App_Msg := 14;

      Start_Build := True;


    when Soft_Reset           =>

      -- Message Format For Group 4 Commands
      --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
      -- 1  | 0   0   0   0   0   0   0   0 |   cmd byte 1 without parity   |
      --    +===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+
      --    | /   /   /   /   /   /   /   /   /   /   /   /   /   /   /   / |
      --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
      --                                    .
      --                                    .
      --                                    .
      --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
      -- 80 |                      num of msg words = 1                     |
      --    +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
      -- Note:  /  represents bits not initialized.
      --         cmd byte 2, 3 and 4 are not used.

      -- Save the command associated with the last application message,
      -- so it can be reported to TM.
      Cmd_Last_App_Msg  := Cmd_Buf(Cmd_buf_index);

      if App_Msg_Counter < App_Msg_Q_Size then

        Q_Tail := (Q_Tail mod App_Msg_Q_Size) + 1;

--SDVS delete (2-dimensional array): App_Msg_Q(Q_Tail, 1) :=
--SDVS delete                                      CONV_BYTE_TO_WORD(Start_Op_Code);
        App_Msg_Q(Q_Tail)(1) := CONV_BYTE_TO_WORD(Start_Op_Code); --SDVS replace

        -- Store application message size for this type of commands
        -- at the end of application message array
--SDVS delete (2-dimensional array): App_Msg_Q(Q_Tail, CMDS_TYPES.App_Msg_Max)
--SDVS delete                                                              := 1;
        App_Msg_Q(Q_Tail)(CMDS_TYPES.App_Msg_Max) := 1; --SDVS replace

        -- XIO RA,DSBL disables int.
--SDVS delete (artclient):       Artclient.Enter_Critical_Section;

        -- *** REMINDER ***
        -- Check assembly to make sure App_Msg_Counter read from RAM
        -- (i.e. current value of App_Msg_Counter) not from register.

        App_Msg_Counter := App_Msg_Counter + 1;
```

```
--SDVS delete (artclient):       Artclient.Leave_Critical_Section;

      else

        -- FOR SOFT RESET, IF THERE IS NOT ROOM, MAKE ROOM!

--SDVS delete (2-dimensional array): App_Msg_Q(Q_Tail, 1) :=
--SDVS delete                                  CONV_BYTE_TO_WORD(Start_Op_Code);
        App_Msg_Q(Q_Tail)(1) := CONV_BYTE_TO_WORD(Start_Op_Code); --SDVS replace


        -- Store application message size for this type of commands
        -- at the end of application message array
--SDVS delete (2-dimensional array): App_Msg_Q(Q_Tail, CMDS_TYPES.App_Msg_Max) := 1;
        App_Msg_Q(Q_Tail)(CMDS_TYPES.App_Msg_Max) := 1;        --SDVS replace
        -- App_Msg_Overwritten_Sys = 15
        TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(15);
        Cmd_Status_Buf(Cmd_buf_index) := App_Msg_Overwritten;

      end if;


    when others =>
      -- Starting_Cmd_Not_Valid = 14
      TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(14);
      Rejected_Cmd_Count :=
             (Rejected_Cmd_Count + 1) mod Max_Rejected_Cmd_Count;
      Cmd_Status_Buf(Cmd_buf_index) := Invalid_Starting_Cmd;

  end case; -- Start_Op_Code



  if Start_Build then

    -- Message Format For Group 1 Commands
    --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    -- 1   | 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 |
    --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    -- 2   | 0   0   0   0   0   0   0   0 |   cmd byte 1 without parity   |
    --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    -- 3   |       cmd byte 2              |           cmd byte 3          |
    --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    -- 4   |       cmd byte 4              |to be filled by continue_build |
    --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    --     |             to be filled by continue build                    |
    --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    --                                     .
    --                                     .
    --                                     .
    --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    --     |             checksum = to be filled by continue build         |
    --     +===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+===+
    --                                          .
    --                                          .
    --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
    -- 80 |        num of msg words = to be filled by continue build      |
    --     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

    App_Msg(1) := 0;
    App_Msg(2) := CONV_BYTE_TO_WORD(Start_Op_Code);

    -- join Second_Byte and Third_Byte and store in App_Msg(3)
    Temp_word      := CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_buf_index).Second_Byte);
    High_Byte_Word := SHIFT_LOGICAL_WORD(Temp_word, 8);

    Temp_Word      := CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_buf_index).Third_Byte);
    App_Msg(3)     := OR_WORD(High_Byte_Word, Temp_Word);

    Temp_word      := CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_buf_index).Fourth_Byte);
    App_Msg(4)     := SHIFT_LOGICAL_WORD(Temp_word, 8);

    -- Save the opcode with parity to be included in Cmd_Last_App_Msg which
    -- will be reported to TM.
    Starting_Cmd_With_Par := Cmd_Buf(Cmd_buf_index).First_Byte;

    -- First of the multiple commands message received.
    Cmds_Rcvd := 1;

    -- Next word to be filled by Continue_Build.
    Word_Cntr            := 4;

    Build_In_Progress    := True;

    Start_Build          := False;

  end if;  -- start_build = true


--SDVS delete (exception):     exception

--SDVS delete (exception):       when others =>

--SDVS delete (exception):         writestring("ibe 2");
--SDVS delete (exception):         writeln;


end INITIATE_BUILD;




-----------------------------------------------------------------------------
--          CONTINUE_BUILD
-----------------------------------------------------------------------------
-- PURPOSE :  This procedure continues to build the application
--            message for the serial digital command whose opcode
--            signals the continuation of the last starting command.
--
-- EXCEPTIONS:
--
-- NOTES:
--
-- CHANGE HISTORY:
--   10/16/91    B. Na     Initial Version
--   mm/dd/yy    B. Na     Build 2 Changes
--
----------------------------------------------------------------------------
procedure CONTINUE_BUILD
          (Cont_Op_Code  : in GLOBAL_TYPES.BYTE;
           Cmd_Buf_Index : in INTEGER) is

  App_Msg_Checksum      : GLOBAL_TYPES.WORD;
  App_Msg_Start_Word    : INTEGER;

  Msg_Word_Index        : INTEGER;
  Q_Word_Index          : INTEGER;

  -- Used to hold intermediate values for bitwise operations.
  Temp_Word             : GLOBAL_TYPES.WORD;
  High_Byte_Word        : GLOBAL_TYPES.WORD;
  Low_Byte_Word         : GLOBAL_TYPES.WORD;

begin

  if Cont_Op_Code = Cont_Cmd then

    -- Continue build the message.

    if (Cmds_Rcvd mod 2) = 0 then

      -- Note: 1st byte of a command is op-code.
      --                               .
      --                               .
      --                               .
      --             +-------------------+---------------------+
      -- Word_Cntr+1 |    2nd byte       |  3rd byte           |
      --             +-------------------+---------------------+
      --          +2 |    4th byte       |(will be filled next)|
      --             +-------------------+---------------------+

      Word_Cntr := Word_Cntr + 1;

      Temp_Word      := CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_Buf_Index).Second_Byte);
      High_Byte_Word := SHIFT_LOGICAL_WORD(Temp_Word, 8);
      Low_Byte_Word  := CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_Buf_Index).Third_Byte);
      App_Msg(Word_Cntr) := OR_WORD(High_Byte_Word, Low_Byte_Word);

      Word_Cntr := Word_Cntr + 1;

      Temp_Word := CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_Buf_Index).Fourth_Byte);
      App_Msg(Word_Cntr) := SHIFT_LOGICAL_WORD(Temp_Word, 8);
      else  -- (Cmds_Rcvd mod 2) /= 0

        -- Note: 1st byte of a command is op-code.
        --                                 .
        --                                 .
        --                                 .
        --          +--------------------+----------------------+
        -- Word_Ctr+0  | (already filled)  |   2nd byte           |
        --          +--------------------+----------------------+
        --         +1  |    3rd byte       |   4th byte           |
        --          +--------------------+----------------------+

        Low_Byte_Word  := CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_Buf_Index).Second_Byte);

        App_Msg(Word_Cntr) := OR_WORD(App_Msg(Word_Cntr), Low_Byte_Word);

        Word_Cntr := Word_Cntr + 1;

        Temp_Word      := CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_Buf_Index).Third_Byte);
        High_Byte_Word := SHIFT_LOGICAL_WORD(Temp_Word, 8);
        Low_Byte_Word  := CONV_BYTE_TO_WORD(Cmd_Buf(Cmd_Buf_Index).Fourth_Byte);
        App_Msg(Word_Cntr) := OR_WORD(High_Byte_Word, Low_Byte_Word);

      end if;

      Cmds_Rcvd := Cmds_Rcvd + 1;

      if Cmds_Rcvd = Num_Cmds_For_App_Msg then

        -- 1)Check checksum (i.e. validate msg).
        -- 2)If good checksum then:
        --       a) If room in Queue, then store msg in App_Msg_Q.
        --       b) Else, report system error and flag app_msg_dropped.
        -- 3)Elseif bad checksum report with cmd status.
        -- 4)Reset build_in_progress flag.

        App_Msg_Checksum := 0;

       -- To verify the checksum, bytes are stored in App_Msg with App_Msg(1)
       -- blank, App_Msg(2) containing the Data op-code for the first cmd
       -- associated with the App_Msg.  Data associated with the checksum
       -- for the App_Msg starts at App_Msg(3).  Num_Words_For_App_Msg does
       -- not include the checksum location.

        for Msg_Word_Index in 3..Num_Words_For_App_Msg loop
          App_Msg_Checksum := XOR_WORD          (App_Msg_Checksum,
                                          App_Msg(Msg_Word_Index));
        end loop;
--@ after_checksum_calculation --SDVS add

        if App_Msg_Checksum = App_Msg(Num_Words_For_App_Msg + 1) then

          -- Checksum good!
        if App_Msg_Counter < App_Msg_Q_Size then

          -- Queue is not full.

          App_Msg_Start_Word := 2;

          if (CONV_WORD_TO_BYTE(App_Msg(2)) = RAM_Mem_Load) or
             (CONV_WORD_TO_BYTE(App_Msg(2)) = EEPROM_Mem_Load) or
             (CONV_WORD_TO_BYTE(App_Msg(2)) = Mem_Dump_X_Frames) then

          -- If memory load or dump then convert
          --
          --     (from this format)      to       (this format)
          --   +----------+---------+          +----------+---------+
          -- 1 |    0     |    0    |          |    0     | opcode  |
          --   +----------+---------+          +----------+---------+
          -- 2 |    0     | opcode  |          |    0     | 2nd byte|
          --   +----------+---------+          +----------+---------+
          -- 3 | 2nd byte | 3rd byte|          |    0     | 3rd byte|
          --   +----------+---------+          +----------+---------+
          -- 4 | 4th byte |         |          | 4th byte |         |
          --   +----------+---------+          +----------+---------+

            App_Msg(1) := App_Msg(2);

--SDVS delete (hex):           High_Byte_Word := AND_WORD(App_Msg(3),16#FF00#);
            High_Byte_Word := AND_WORD(App_Msg(3),65280); --SDVS replace

            App_Msg(2) := SHIFT_LOGICAL_WORD(High_Byte_Word, -8);

--SDVS delete (hex):           Low_Byte_Word := AND_WORD(App_Msg(3),16#00FF#);
            Low_Byte_Word := AND_WORD(App_Msg(3),255); --SDVS replace

            App_Msg(3) := Low_Byte_Word;

            App_Msg_Start_Word := 1;

          end if;

          -- The message has been built.  Now store it in the queue
          -- if the queue is not full.

          Q_Tail := (Q_Tail mod App_Msg_Q_Size) + 1;

          -- To include checksum at the end of the message
          Num_Words_For_App_Msg := Num_Words_For_App_Msg + 1;

          Q_Word_Index := 0;

          -- Copy from message buffer to queue
          for Msg_Word_Index in App_Msg_Start_Word..Num_Words_For_App_Msg loop
            Q_Word_Index := Q_Word_Index + 1;
--SDVS delete (2-dimensional array):  App_Msg_Q(Q_Tail, Q_Word_Index) :=
--SDVS delete                                           App_Msg(Msg_Word_Index);
            App_Msg_Q(Q_Tail)(Q_Word_Index) :=
                                      App_Msg(Msg_Word_Index); --SDVS replace
          end loop;

          -- Store the size of this application message at the bottom
          -- so that MANAGE_MSG_RETRIEVAL can retrieve the exact number
          -- of words for this message.
--SDVS delete (1750A)  MEMORY_MANAGER.CONV_TYPE
--SDVS delete (1750A)        (Q_Word_Index'address,                               -- Source
--SDVS delete (1750A     App_Msg_Q(Q_Tail, CMDS_TYPES.App_Msg_Max)'address, -- Target
--SDVS delete (1750A       1);                                                -- Words
               App_Msg_Q(Q_Tail)(CMDS_TYPES.App_Msg_Max) := --SDVS replace
                    GLOBAL_TYPES.WORD(Q_Word_Index); --SDVS replace

          -- XIO RA,DSBL disables interrupt.  Programmers should ensure
          -- that exceptions do not cause calls to Leave_Critical_Section
          -- to be missed. Refer to page 8-21 of the Tartan manual.
--SDVS delete (artclient):          Artclient.Enter_Critical_Section;

          -- *** REMINDER ***
          -- Check assembly to make sure App_Msg_Counter read from RAM
          -- (i.e. current value of App_Msg_Counter) not from register.

          App_Msg_Counter := App_Msg_Counter + 1;

          -- XIO RA,ENBL enables int.
--SDVS delete (artclient):          Artclient.Leave_Critical_Section;

          -- Save the latest start cmd that identifies the type of
          -- applicaton message last received.  This will be collected
          -- by TM.COLLECT during tracking state.
          --
          -- App_Msg Format (shown only first 3 1/2 words)
          --                                 for the following cmds:
          --                                     RAM_Mem_Load,
          --                                     EEPROM_Mem_Load,
          --      for other cmds                 Mem_Dump_X_Frames.
          --    +----------+---------+         +----------+---------+
          -- 1 |     0    |    0    |         |     0    | opcode  |
          --    +----------+---------+         +----------+---------+
          -- 2 |     0    | opcode  |         |     0    | 2nd byte|
          --    +----------+---------+         +----------+---------+
          -- 3 | 2nd byte | 3rd byte|         |     0    | 3rd byte|
          --    +----------+---------+         +----------+---------+
          -- 4 | 4th byte |         |         | 4th byte |         |
          --    +----------+---------+         +----------+---------+

          Cmd_Last_App_Msg.First_Byte := Starting_Cmd_With_Par;

          if App_Msg_Start_Word = 1 then  -- For memory load or dump message
            -- Second_Byte is App_Msg(2).Low_Byte;
            Cmd_Last_App_Msg.Second_Byte := CONV_WORD_TO_BYTE(App_Msg(2));
          else  -- All other message
            -- Second_Byte is App_Msg(2).High_Byte;
            Temp_Word := SHIFT_LOGICAL_WORD(App_Msg(3), -8);
            Cmd_Last_App_Msg.Second_Byte := CONV_WORD_TO_BYTE(Temp_Word);
          end if;

          --  Third_Byte is App_Msg(3).Low_Byte;
--SDVS comment: The assignment statement that follows can not be executed in SDVS
--SDVS comment: (the result of its execution  is compiler depended), because
--SDVS comment: the result of using unchecked conversion to convert App_Msg(3),
--SDVS comment: which is a "word" with value possibly greater than 255, to a
--SDVS comment: byte is unknown. This is a safeguard that SDVS places on unchecked
--SDVS comment: conversions. The authors of the MSX code are assuming that their
--SDVS comment: compiler will execute the assignment statement by assigning to
--SDVS comment: Cmd_Last_App_Msg.Third_Byte the lower byte of App_Msg(3).

--SDVS delete:          Cmd_Last_App_Msg.Third_Byte := CONV_WORD_TO_BYTE(App_Msg(3));
          Cmd_Last_App_Msg.Third_Byte :=
                          CONV_WORD_TO_BYTE(AND_WORD(App_Msg(3),255)); --SDVS replace

          --  Fourth_Byte is App_Msg(4).High_Byte;
          Temp_Word := SHIFT_LOGICAL_WORD(App_Msg(4), -8);
          Cmd_Last_App_Msg.Fourth_Byte := CONV_WORD_TO_BYTE(Temp_Word);

        else

          -- gets time-tagged in TM_Data
          -- App_Msg_Q_Overflowed = 17
          TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(17);
          Cmd_Status_Buf(Cmd_Buf_Index) := App_Msg_Dropped;

          Rejected_Cmd_Count :=
               (Rejected_Cmd_Count + 1) mod Max_Rejected_Cmd_Count;

        end if;

      else  -- Checksum failed
        -- Checksum_Failed = 18
        TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(18);

        Rejected_Cmd_Count :=
             (Rejected_Cmd_Count + 1) mod Max_Rejected_Cmd_Count;
        Cmd_Status_Buf(Cmd_Buf_Index) := Bad_Checksum;
      end if; -- App_Msg_Checksum good or bad

      Build_In_Progress := False;

    end if;  -- Cmds_Rcvd = Num_Cmds_For_App_Msg

  else -- if Cmd_Buf(Cmd_Buf_Index).Cont_Op_Code /= Cont_Cmd then

    -- The No-op cmd could be interspersed in another cmd. This
    -- is because of the prioritized order of cmds from the TP
    -- being higher than delayed cmds.  This is the only cmd that
    -- can be interspersed.

    if (Cont_Op_Code = No_op_Cmd) then

      if (App_Msg_Counter < App_Msg_Q_Size) then
```

```
        Q_Tail := (Q_Tail mod App_Msg_Q_Size) + 1;

--SDVS delete (2-dimensional array): App_Msg_Q(Q_Tail, 1) :=
--SDVS delete                                CONV_BYTE_TO_WORD(Cont_Op_Code);
        App_Msg_Q(Q_Tail)(1) := CONV_BYTE_TO_WORD(Cont_Op_Code); --SDVS replace

        -- store application message size for this type of commands
        -- at the end of application message array
--SDVS delete (2-dimensional array): App_Msg_Q(Q_Tail, CMDS_TYPES.App_Msg_Max)
--SDVS delete                                                                  := 1;
        App_Msg_Q(Q_Tail)(CMDS_TYPES.App_Msg_Max) := 1;        --SDVS replace

        -- XIO RA,DSBL disables int.
--SDVS delete (artclient):       Artclient.Enter_Critical_Section;

        -- *** REMINDER ***
        -- Check assembly to make sure App_Msg_Counter read from RAM
        -- (i.e. current value of App_Msg_Counter) not from register.

        App_Msg_Counter := App_Msg_Counter + 1;

        -- XIO RA,ENBL enables int.
--SDVS delete (artclient):       Artclient.Leave_Critical_Section;

        Cmd_Last_App_Msg := Cmd_Buf(Cmd_Buf_Index);
      else
        -- App_Msg_Q_Overflowed = 17
        TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(17);
        Cmd_Status_Buf(Cmd_Buf_Index) := App_Msg_Dropped;

        Rejected_Cmd_Count :=
             (Rejected_Cmd_Count + 1) mod Max_Rejected_Cmd_Count;
      end if;

    else  -- Cont_Op_Code is not No_Op_Cmd
      -- Prior_Build_Flushed = 12
      TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(12);
      Rejected_Cmd_Count :=
          (Rejected_Cmd_Count + 1) mod Max_Rejected_Cmd_Count;
      Cmd_Status_Buf(Cmd_Buf_Index) := Flushed_Prior_Build;

      Build_In_Progress := False;

    end if;

  end if;  -- build in progress and next_cmd = or /= Cont_Cmd


--SDVS delete (exception):      exception

--SDVS delete (exception):        when others =>

--SDVS delete (exception):         writestring("cbe 2");
--SDVS delete (exception):          writeln;
```

```
end CONTINUE_BUILD;

--*****************************************************************************

begin

--SDVS delete (TM):  for I in 1..Latest_Cmds_Q_Size loop

--SDVS delete (TM):    Latest_Cmds_Q(I) := Zero_Cmd;

--SDVS delete (TM):  end loop;

  null;

--SDVS delete (exception): exception

--SDVS delete (exception):  when tasking_error =>

--SDVS delete (exception):    writestring("apte 5");
--SDVS delete (exception):    writeln;
    -- App_Msgs_Pkg_Tasking_Exception = 76
--SDVS delete (exception):    TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(76);

--SDVS delete (exception):  when others =>

--SDVS delete (exception):    writestring("ape 6");
--SDVS delete (exception):    writeln;
    -- App_Msgs_Pkg_Exception = 77
--SDVS delete (exception):    TM_DATA.MANAGE.REPORT_SYSTEM_ERROR(77);


end APP_MSGS;


begin --SDVS add
 --@ loop_begin --SDVS add

  loop --SDVS add
     SERIAL_DIG_CMDS.CMD_IN_HANDLER; --SDVS add
     for i in  2 .. APP_MSGS.RET_MSG_LENGTH(SERIAL_DIG_CMDS.RET_MSG_ID) loop --SDVS add
        SERIAL_DIG_CMDS.CMD_IN_HANDLER; --SDVS add
     end loop; --SDVS add
     APP_MSGS.BUILD; --SDVS add
     APP_MSGS.PROCESS_MSG; --SDVS add
  end loop; --SDVS add

end MSX_PROGRAM_FINAL_VERSION; --SDVS add
```

9 Appendix B – The Proofs

9.1 Definitions, etc. used in the main proof

```
; -*- Syntax: Common-lisp; Package: USER; Mode: LISP -*-~% ; File name
: /u/versys/msx/infinite_sequence_data_structure_messages.defs

;;; This file contains the formulas, macros and proofs used in the main proof
;;; (found in file
;;; /u/versys/msx/infinite_sequence_data_structure_messages.proof).
;;; This file is for the most part organized bottom-up: that is, definitions,
;;; lemmas, etc. are introduced immediately before they are used. The major
;;; exception to this is the first section which contains macros. A top-down
;;; explanation of main proof is given in the body of the technical report.
```

9.1.1 Macros

```
;;; Macro section

;; Macro is.byte is a predicate which is true if x is between 0 and 255
;; inclusive, false otherwise.

(defmacroo is.byte
   "0 le x & x le 255"
   (x) nil)

;; Macro is.byte is a predicate which is true if x is between 0 and 65536
;;inclusive, false otherwise.

(defmacroo is.word
   "0 le x & x le 65535"
  (x) nil)

;; Macro mk.word is a function which returns the value of a word whose
;; high-order bits are the byte a and whose low-order bits are the byte b.

(defmacroo mk.word
   "256 * a + b"
  (a b) nil)

;; Macro all.high is a predicate which returns true if x is a word all of
;; whose low-order bits are 0, false otherwise.

(defmacroo all.high
    "0 le x & x le 65535 & x mod 256 = 0" ; The is.word macro is not used
  (x) nil)                                ; here as SDVS does not currently
                                          ; handle nested macros

;; Macro all.low is a predicate which returns true if x is a word all of whose
;; low-order bits are 0, false otherwise.

(defmacroo all.low
   "0 le x & x le 255"
  (x) nil)

;; Macro high.bits is a function whose value is the high-order bits of a word.

(defmacroo high.bits
    "x / 256"
  (x) nil)

;; Macro low.bits is a function whose value is the low-order bits of a word.
(defmacroo low.bits
    "x mod 256"
  (x) nil)
```

9.1.2 Timing proofs

```
;;; Proofs used for timing. These proofs are used to give the date at various points
;;; in the main proof if the variable timing is set to true.

;; Proof printdate prints the date if the timing flag is true, does nothing otherwise.

(defproof printdate_if_wanted
    "(if timing
         then printdate)")

;;Proof printdate simply prints the date

(defproof printdate
    "(date)")
```

9.1.3 Formulas used in the precondition

```
;;; The following section contains formulas used in the precondition
;;; of the main state delta (infinite_sequence_data_structure_messages.sd).

;;; The following formulas (which are accompanied below by their
;;; proofs) are included in the precondition to avoid re-proving for
;;; each run with a different step_case.proof
;;; The state deltas input_places_disjoint.sd and
;;; output_places_disjoint.sd are used in the step case to prove
;;; that the input and output places associated with the nth message are
;;; disjoint from the places mentioned in the output condition (i.e., those
;;; associated with the xth message).

;; State delta input_places_disjoint.sd is applied in the n lt x case in the
;; step case of the induction so that the prover is able to deduce that the input
;; places for the nth message are disjoint from the input places for the xth message.

(defsd input_places_disjoint.sd
   "[sd pre: ( formula(msg_input_begins_at_definition),n ge 1, n lt x,
              formula(msg_in_lh_definition),formula(allowed_message_ids))
               post: (msg_input_begins_at(n) + msg_in_lh(n) - 1
                       lt msg_input_begins_at(x))]")

;; Proof input_places_disjoint.proof is the proof of input_places_disjoint.sd.

(defproof input_places_disjoint.proof
    "(setflag traceflag on,
      comment \ "Beginning interpretation of input_places_disjoint.proof... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

     ; The proof is a fairly straightforward induction on n - x

      natural induction on: k
                formulas:    (forall x ((n ge 1 & n lt x) &
                                         k = x - (n + 1)
                                         --> msg_input_begins_at(n)
                                              + (msg_in_lh(n)-1)
                                               lt msg_input_begins_at((n +
                                                                           k) +
                                                                          1)))
                base proof:
                  ( provebyinstantiation
                     using: msg_in_lh_definition
                     substitutions: (x=n),
                   provebyinstantiation
                     using: msg_input_begins_at_definition
                     substitutions: (x=n + 1),
                   close)

                step proof:
                  (letq induct_hyp = q(1),
                   provebyinstantiation
                     using: msg_input_begins_at_definition
                     substitutions: (x=(n + k) + 2),
                   provebyinstantiation
                     using: msg_in_lh_definition
                     substitutions: (x=(n + k) + 1),
                   provebyinstantiation
                     using: induct_hyp
                     substitutions: (x=(n + k) + 1),
                   provebyinstantiation
                     using: allowed_message_ids
                     substitutions: (x=(n + k) + 1),
              close),

              provebyinstantiation
                using: q(1)
                substitutions: (k=x - (n + 1)),
              enotice x - (n + 1) ge 0,
              notice
               x - (n + 1) ge 0
                  --> forall x ((n ge 1 & n lt x) &
                                  x - (n + 1) = x - (n + 1)
                                  --> msg_input_begins_at(n) + (msg_in_lh(n)-1)
                                         lt msg_input_begins_at((n +
                                                                    (x - (n +
                                                                            1))) +
                                                                  1)),
              notice
               forall x ((n ge 1 & n lt x) &
                           x - (n + 1) = x - (n + 1)
                           --> msg_input_begins_at(n) + (msg_in_lh(n)-1)
                                  lt msg_input_begins_at((n +
                                                             (x - (n + 1))) +
                                                           1)),
              provebyinstantiation
                using: q(1)
                substitutions: (x=x),

              interpret printdate_if_wanted,

              setflag traceflag on,
              comment
                \ "Finished with interpretation of input_places_disjoint.proof... \ ",
              setflag traceflag off,
              close)")

;; Proof separate_input_places simply applies input_places_disjoint.sd.

(defproof separate_input_places
    "(setflag traceflag on,
      comment  \ "Beginning interpretation of separate_input_places... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      apply input_places_disjoint.sd,

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of separate_input_places... \ ",
      setflag traceflag off)")

;; State delta output_places_disjoint.sd is applied in the n lt x case in the
;; step case of the induction so that the prover is able to deduce that the
;; output places for the nth message are disjoint from the output places for
;; the xth message.

(defsd output_places_disjoint.sd
   "[sd pre: ( formula(msg_output_begins_at_definition),n ge 1, n lt x,
              formula(msg_out_lh_definition),formula(allowed_message_ids))
               post: (msg_output_begins_at(n) + msg_out_lh(n) - 1
                       lt msg_output_begins_at(x))]")

;; Proof output_places_disjoint.proof is the proof of output_places_disjoint.sd.

(defproof output_places_disjoint.proof
    "(setflag traceflag on,
      comment  \ "Beginning interpretation of output_places_disjoint.proof... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      ; Again, the proof is a fairly straightforward induction on n - x

      natural induction on: k
                formulas:    (forall x ((n ge 1 & n lt x) &
                                         k = x - (n + 1)
                                         --> msg_output_begins_at(n)
                                              + (msg_out_lh(n)-1)
                                                lt msg_output_begins_at((n +
                                                                         k) +
                                                                       1)))
                base proof:
                  ( provebyinstantiation
                     using: msg_out_lh_definition
                     substitutions: (x=n),
                   provebyinstantiation
                     using: msg_output_begins_at_definition
                     substitutions: (x=n + 1),
                   close)

                step proof:
                  (letq induct_hyp = q(1),
                   provebyinstantiation
                     using: msg_output_begins_at_definition
                     substitutions: (x=(n + k) + 2),
                   provebyinstantiation
                     using: msg_out_lh_definition
                     substitutions: (x=(n + k) + 1),
                   provebyinstantiation
                     using: induct_hyp
                     substitutions: (x=(n + k) + 1),
                   provebyinstantiation
                     using: allowed_message_ids
                     substitutions: (x=(n + k) + 1),
                   close),

              provebyinstantiation
```

```
                  using: q(1)
                  substitutions: (k=x - (n + 1)),
                enotice x - (n + 1) ge 0,
                notice
                 x - (n + 1) ge 0
                    --> forall x ((n ge 1 & n lt x) &
                                   x - (n + 1) = x - (n + 1)
                                   --> msg_output_begins_at(n) + (msg_out_lh(n)-1)
                                           lt msg_output_begins_at((n +
                                                                    (x - (n +
                                                                            1))) +
                                                                   1)),
                notice
                 forall x ((n ge 1 & n lt x) &
                           x - (n + 1) = x - (n + 1)
                           --> msg_output_begins_at(n) + (msg_out_lh(n)-1)
                                   lt msg_output_begins_at((n +
                                                            (x - (n + 1))) +
                                                           1)),
                provebyinstantiation
                  using: q(1)
                  substitutions: (x=x),

                interpret printdate_if_wanted,

                setflag traceflag on,
                comment
                  \ "Finished with interpretation of output_places_disjoint.proof... \ ",
                setflag traceflag off,
                close)")

;; Proof separate_input_places simply applies input_places_disjoint.sd.

(defproof separate_output_places
    "(setflag traceflag on,
      comment  \ "Beginning interpretation of separate_output_places... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      apply output_places_disjoint.sd,

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of separate_output_places... \ ",
      setflag traceflag off)")

;; State delta finale.sd is used after the induction and allows the main
;; proof to be run without using EKL.

(defsd finale.sd
   "[sd pre: (x + 1 = x + 1
               --> (x ge 1
                   --> forall z (1 le z & z le msg_out_lh(x)
                                          --> .stdout
                                                [msg_output_begins_at(x) +
                                                   (z - 1)]
                                                 = mk.word
                                                    (.stdin
                                                      [msg_input_begins_at(x) +
                                                        (8 * z - 5) / 3],
                                                     .stdin
                                                       [msg_input_begins_at(x) +
                                                         (8 * z - 1) / 3])))
       post: (x ge 1
                 --> forall z (1 le z & z le msg_out_lh(x)
                                 --> .stdout
                                       [msg_output_begins_at(x) + (z - 1)]
                                        = mk.word
                                           (.stdin
                                              [msg_input_begins_at(x) +
                                                 (8 * z - 5) / 3],
                                            .stdin
                                              [msg_input_begins_at(x) +
                                                 (8 * z - 1) / 3]))))]")

;; Proof finale.proof is the (completely automatic) proof of finale.sd.

(defproof finale.proof
   "(quantification on,
     prove finale.sd
       proof: )")

;;; The rest of the formulas in this section specify that the input is
;;; correct.

;; Formula allowed_message_ids states which message ids are allowed (i.e., are
;; actually legitimate message ids for data-structure messages.


(defformula allowed_message_ids
    "forall x ((x ge 1) --> msg_id(x) = 1
                         or msg_id(x) = 2
                         or msg_id(x) = 3
                         or msg_id(x) = 4
                         or msg_id(x) = 5
                         or msg_id(x) = 6
                         or msg_id(x) = 7
                         or msg_id(x) = 8
                         or msg_id(x) = 9
                         or msg_id(x) = 10
                         or msg_id(x) = 11
                         or msg_id(x) = 12
                         or msg_id(x) = 13
                         or msg_id(x) = 14
                         or msg_id(x) = 15
                         or msg_id(x) = 16
                         or msg_id(x) = 17
                         or msg_id(x) = 18
```

```
                        or msg_id(x) = 19
                        or msg_id(x) = 20
                        or msg_id(x) = 21
                        or msg_id(x) = 22
                        or msg_id(x) = 23
                        or msg_id(x) = 24
                        or msg_id(x) = 25
                        or msg_id(x) = 26
                        or msg_id(x) = 27
                        or msg_id(x) = 28
                        or msg_id(x) = 29
                        or msg_id(x) = 50
                        or msg_id(x) = 51
                        or msg_id(x) = 52
                        or msg_id(x) = 53
                        or msg_id(x) = 54
                        or msg_id(x) = 55
                        or msg_id(x) = 56
                        or msg_id(x) = 57
                        or msg_id(x) = 58
                        or msg_id(x) = 59
                        or msg_id(x) = 60
                        or msg_id(x) = 61
                        or msg_id(x) = 62
                        or msg_id(x) = 63
                        or msg_id(x) = 64
                        or msg_id(x) = 65
                        or msg_id(x) = 66
                        or msg_id(x) = 67
                        or msg_id(x) = 80
                        or msg_id(x) = 81
                        or msg_id(x) = 82
                        or msg_id(x) = 83
                        or msg_id(x) = 84
                        or msg_id(x) = 85
                        or msg_id(x) = 86
                        or msg_id(x) = 87
                        or msg_id(x) = 88
                        or msg_id(x) = 89
                        or msg_id(x) = 90
                        or msg_id(x) = 91
                        or msg_id(x) = 92
                        or msg_id(x) = 93)")


;; Formula msg_in_lh_definition defines the function msg_in_lh.
;; This function gives the number of bytes of input for a message. N.B: The values
;; of this function are four times as large as the values in the
;; specification, as there the number of commands rather than the number of
;; bytes is used.


(defformula msg_in_lh_definition
    "forall x ((x gt 0) -->
      (if msg_id(x) = 1  then msg_in_lh(x) = 56 else
(if msg_id(x) = 2  then msg_in_lh(x) = 56 else
(if msg_id(x) = 3  then msg_in_lh(x) = 56 else
(if msg_id(x) = 4  then msg_in_lh(x) = 56 else
(if msg_id(x) = 5  then msg_in_lh(x) = 56 else
(if msg_id(x) = 6  then msg_in_lh(x) = 56 else
(if msg_id(x) = 7  then msg_in_lh(x) = 56 else
(if msg_id(x) = 8  then msg_in_lh(x) = 40 else
(if msg_id(x) = 9  then msg_in_lh(x) = 40 else
(if msg_id(x) = 10 then msg_in_lh(x) = 12 else
(if msg_id(x) = 11 then msg_in_lh(x) = 12 else
(if msg_id(x) = 12 then msg_in_lh(x) = 16 else
(if msg_id(x) = 13 then msg_in_lh(x) = 16 else
(if msg_id(x) = 14 then msg_in_lh(x) = 12 else
(if msg_id(x) = 15 then msg_in_lh(x) = 12 else
(if msg_id(x) = 16 then msg_in_lh(x) = 12 else
(if msg_id(x) = 17 then msg_in_lh(x) = 12 else
(if msg_id(x) = 18 then msg_in_lh(x) = 12 else
(if msg_id(x) = 19 then msg_in_lh(x) = 32 else
(if msg_id(x) = 20 then msg_in_lh(x) = 12 else
(if msg_id(x) = 21 then msg_in_lh(x) = 12 else
(if msg_id(x) = 22 then msg_in_lh(x) = 8 else
(if msg_id(x) = 23 then msg_in_lh(x) = 16 else
(if msg_id(x) = 24 then msg_in_lh(x) = 8 else
(if msg_id(x) = 25 then msg_in_lh(x) = 16 else
(if msg_id(x) = 26 then msg_in_lh(x) = 16 else
(if msg_id(x) = 27 then msg_in_lh(x) = 16 else
(if msg_id(x) = 28 then msg_in_lh(x) = 8 else
(if msg_id(x) = 29 then msg_in_lh(x) = 24 else
(if msg_id(x) = 50 then msg_in_lh(x) = 40 else
(if msg_id(x) = 51 then msg_in_lh(x) = 16 else
(if msg_id(x) = 52 then msg_in_lh(x) = 24 else
(if msg_id(x) = 53 then msg_in_lh(x) = 152 else
(if msg_id(x) = 54 then msg_in_lh(x) = 152 else
(if msg_id(x) = 55 then msg_in_lh(x) = 208 else
(if msg_id(x) = 56 then msg_in_lh(x) = 208 else
(if msg_id(x) = 57 then msg_in_lh(x) = 52 else
(if msg_id(x) = 58 then msg_in_lh(x) = 24 else
(if msg_id(x) = 59 then msg_in_lh(x) = 16 else
(if msg_id(x) = 60 then msg_in_lh(x) = 24 else
(if msg_id(x) = 61 then msg_in_lh(x) = 16 else
(if msg_id(x) = 62 then msg_in_lh(x) = 20 else
(if msg_id(x) = 63 then msg_in_lh(x) = 44 else
(if msg_id(x) = 64 then msg_in_lh(x) = 28 else
(if msg_id(x) = 65 then msg_in_lh(x) = 32 else
(if msg_id(x) = 66 then msg_in_lh(x) = 32 else
(if msg_id(x) = 67 then msg_in_lh(x) = 60 else
(if msg_id(x) = 80 then msg_in_lh(x) = 16 else
(if msg_id(x) = 81 then msg_in_lh(x) =  8 else
(if msg_id(x) = 82 then msg_in_lh(x) =  8 else
(if msg_id(x) = 83 then msg_in_lh(x) =  8 else
(if msg_id(x) = 84 then msg_in_lh(x) =  8 else
(if msg_id(x) = 85 then msg_in_lh(x) =  8 else
(if msg_id(x) = 86 then msg_in_lh(x) = 16 else
(if msg_id(x) = 87 then msg_in_lh(x) = 16 else
(if msg_id(x) = 88 then msg_in_lh(x) = 16 else
      (if msg_id(x) = 89 then msg_in_lh(x) = 16 else
      (if msg_id(x) = 90 then msg_in_lh(x) =  8 else
      (if msg_id(x) = 91 then msg_in_lh(x) = 44 else
      (if msg_id(x) = 92 then msg_in_lh(x) = 28 else
      (if msg_id(x) = 93 then msg_in_lh(x) =  8 else msg_in_lh(x) = 0
       ))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))")

;; Formula msg_out_lh_definition defines the function msg_out_lh.
;; This function gives the number of words of output for a message.


(defformula msg_out_lh_definition
    "forall x ((x gt 0) -->
      (if msg_id(x) = 1 then msg_out_lh(x) = 20 else
      (if msg_id(x) = 2 then msg_out_lh(x) = 20 else
      (if msg_id(x) = 3 then msg_out_lh(x) = 20 else
      (if msg_id(x) = 4 then msg_out_lh(x) = 20 else
      (if msg_id(x) = 5 then msg_out_lh(x) = 20 else
      (if msg_id(x) = 6 then msg_out_lh(x) = 20 else
      (if msg_id(x) = 7 then msg_out_lh(x) = 20 else
      (if msg_id(x) = 8 then msg_out_lh(x) = 14 else
      (if msg_id(x) = 9 then msg_out_lh(x) = 14 else
      (if msg_id(x) = 10 then msg_out_lh(x) = 4 else
      (if msg_id(x) = 11 then msg_out_lh(x) = 4 else
      (if msg_id(x) = 12 then msg_out_lh(x) = 6 else
      (if msg_id(x) = 13 then msg_out_lh(x) = 6 else
      (if msg_id(x) = 14 then msg_out_lh(x) = 4 else
      (if msg_id(x) = 15 then msg_out_lh(x) = 4 else
      (if msg_id(x) = 16 then msg_out_lh(x) = 4 else
      (if msg_id(x) = 17 then msg_out_lh(x) = 4 else
      (if msg_id(x) = 18 then msg_out_lh(x) = 4 else
      (if msg_id(x) = 19 then msg_out_lh(x) = 11 else
      (if msg_id(x) = 20 then msg_out_lh(x) = 4 else
      (if msg_id(x) = 21 then msg_out_lh(x) = 4 else
      (if msg_id(x) = 22 then msg_out_lh(x) = 3 else
      (if msg_id(x) = 23 then msg_out_lh(x) = 6 else
      (if msg_id(x) = 24 then msg_out_lh(x) = 3 else
      (if msg_id(x) = 25 then msg_out_lh(x) = 5 else
      (if msg_id(x) = 26 then msg_out_lh(x) = 5 else
      (if msg_id(x) = 27 then msg_out_lh(x) = 6 else
      (if msg_id(x) = 28 then msg_out_lh(x) = 3 else
      (if msg_id(x) = 29 then msg_out_lh(x) = 8 else
      (if msg_id(x) = 50 then msg_out_lh(x) = 15 else
      (if msg_id(x) = 51 then msg_out_lh(x) = 6 else
      (if msg_id(x) = 52 then msg_out_lh(x) = 8 else
      (if msg_id(x) = 53 then msg_out_lh(x) = 57 else
      (if msg_id(x) = 54 then msg_out_lh(x) = 57 else
      (if msg_id(x) = 55 then msg_out_lh(x) = 78 else
      (if msg_id(x) = 56 then msg_out_lh(x) = 78 else
      (if msg_id(x) = 57 then msg_out_lh(x) = 19 else
      (if msg_id(x) = 58 then msg_out_lh(x) = 8 else
      (if msg_id(x) = 59 then msg_out_lh(x) = 6 else
      (if msg_id(x) = 60 then msg_out_lh(x) = 9 else
      (if msg_id(x) = 61 then msg_out_lh(x) = 6 else
      (if msg_id(x) = 62 then msg_out_lh(x) = 7 else
      (if msg_id(x) = 63 then msg_out_lh(x) = 16 else
      (if msg_id(x) = 64 then msg_out_lh(x) = 10 else
      (if msg_id(x) = 65 then msg_out_lh(x) = 11 else
      (if msg_id(x) = 66 then msg_out_lh(x) = 11 else
      (if msg_id(x) = 67 then msg_out_lh(x) = 22 else
      (if msg_id(x) = 80 then msg_out_lh(x) =  5 else
      (if msg_id(x) = 81 then msg_out_lh(x) =  3 else
      (if msg_id(x) = 82 then msg_out_lh(x) =  3 else
      (if msg_id(x) = 83 then msg_out_lh(x) =  3 else
      (if msg_id(x) = 84 then msg_out_lh(x) =  3 else
      (if msg_id(x) = 85 then msg_out_lh(x) =  3 else
      (if msg_id(x) = 86 then msg_out_lh(x) =  5 else
      (if msg_id(x) = 87 then msg_out_lh(x) =  5 else
      (if msg_id(x) = 88 then msg_out_lh(x) =  6 else
      (if msg_id(x) = 89 then msg_out_lh(x) =  5 else
      (if msg_id(x) = 90 then msg_out_lh(x) =  3 else
      (if msg_id(x) = 91 then msg_out_lh(x) = 16 else
      (if msg_id(x) = 92 then msg_out_lh(x) = 10 else
      (if msg_id(x) = 93 then msg_out_lh(x) =  3 else msg_out_lh(x) = 0
       ))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))")

;; Formula msg_input_begins_at_definition defines the
;; function msg_input_begins_at which gives the beginning input location
;; for messages.

(defformula msg_input_begins_at_definition
    "forall x (msg_input_begins_at(1) = 1 &
              ((x gt 1) -->
                (msg_input_begins_at(x) =
                  msg_input_begins_at(x-1) + msg_in_lh(x-1))))")

;; Formula msg_output_begins_at_definition defines the
;; function msg_output_begins_at which gives the beginning output location
;; for messages.


(defformula msg_output_begins_at_definition
    "forall x (msg_output_begins_at(1) = 1 &
              ((x gt 1) -->
                (msg_output_begins_at(x) =
                  msg_output_begins_at(x-1) + msg_out_lh(x-1))))")

;; Formula checksum_first_k_bytes_message_n_definition defines the function
;; checksum_first_k_bytes_message_n(n,k) , which gives the result of applying
;; the sdvs_xor_word function to the first k actual input bytes of message n.

(defformula checksum_first_k_bytes_message_n_definition
    "forall n
      (forall k (
          (k = 0  -->
              checksum_first_k_bytes_message_n(n,k) = 0) &
          (k gt 0 -->
              (checksum_first_k_bytes_message_n(n,k)
                 = sdvs_xor_word(checksum_first_k_bytes_message_n(n,k-1),
                    mk.word(.stdin[msg_input_begins_at(n) + ((8 * k - 5) / 3)],
                              .stdin[msg_input_begins_at(n) + ((8 * k - 1) / 3)]))))))")

;; Formula checksum_definition defines the function checksum(x), which gives
;; the result of applying the sdvs_xor_word function to all of the actual
;; input bytes of message x.


(defformula checksum_definition
    "(forall x (x ge 0 -->
            (checksum(x) = checksum_first_k_bytes_message_n(x, msg_out_lh(x) -1))
            & is.word(checksum(x))))")

;; Formula input_condition is the principal formula used to specify the input.

(defformula input_condition
    "forall x
      (forall z
       (x ge 1 -->

          ; The first byte is always 1.
            ((z = 1 --> .stdin[msg_input_begins_at(x) + (z-1)] = 1) &

          ; The second byte is always 2.
            (z = 2 --> .stdin[msg_input_begins_at(x) + (z-1)] = 2) &

          ; The third byte is the message identifier.
            (z = 3 --> .stdin[msg_input_begins_at(x) + (z-1)] = msg_id(x)) &

          ; As for the remaining bytes...
            ((4 le z & z le  msg_in_lh(x)) -->

              ; If it's the first byte of a command, it's an eight:
                ((z mod 4 = 1 --> .stdin[msg_input_begins_at(x) + (z-1)] = 8) &

              ; otherwise, we only know it's a byte.
                (z mod 4 ~= 1 --> is.byte(.stdin[msg_input_begins_at(x) + (z-1)])))) &

          ; First checksum byte is the high bits of the checksum word for message x
              ((z =  ((8 * msg_out_lh(x) - 2) / 3)) -->
                      .stdin[msg_input_begins_at(x) + (z-1)] =
                          high.bits(checksum(x))) &

          ; Second checksum byte is the low bits of the checksum word for message x
              ((z =  ((8 * msg_out_lh(x) + 2) / 3)) -->
                        .stdin[msg_input_begins_at(x) + (z-1)] =
                          low.bits(checksum(x))))))")
```

9.1.4 Formulas used in the postcondition

```
;;; Formulas used in the postcondition of the main state delta
;;; (infinite_sequence_data_structure_messages.sd)
```

```
;; Formula output_condition states that all words output for the xth message are
;; correct, i.e., are words formed from the specified inputs in the high and low bytes.


(defformula output_condition
    "x ge 1 --> (forall  z (( 1 le z & z le msg_out_lh(x))
                --> #stdout[msg_output_begins_at(x) +(z-1)]
                       = mk.word(.stdin[msg_input_begins_at(x) + ((8 * z - 5) / 3)],
                                  .stdin[msg_input_begins_at(x) + ((8 * z - 1) / 3)])))")
```

9.1.5 The correctness assertion

```
;;; The correctness assertion

;; State delta infinite_sequence_data_structure_messages.sd is the correctness
;; assertion and states that if the input is correct, the program eventually
;; produces the correct output.


(defsd infinite_sequence_data_structure_messages.sd
    "[sd pre: (formula(finale.sd),
               formula(input_places_disjoint.sd),
               formula(output_places_disjoint.sd),
               ada(msx_program_final_version.a),
               formula(checksum_first_k_bytes_message_n_definition),
               formula(checksum_definition),
               formula(msg_in_lh_definition),
               formula(msg_out_lh_definition),
               formula(msg_input_begins_at_definition),
               formula(msg_output_begins_at_definition),
               formula(allowed_message_ids),
               formula(input_condition))
      comod: (all)
        mod: (all)
       post: (formula(output_condition)) ]")
```

9.1.6 Proofs used in the main proof

```
;;; Proofs used in the main proof. Each of these proofs is embedded
;;; in setflag and comment commands that allow the traceflag to be turned off
;;; in the main proof so that an abbrieviated trace can be obtained. Each
;;; proof is also surrounded by printdate_if_wanted commands that allow timing
;;; information to be obtained if the timing variable is set to true in the
;;; main proof.
;;; The following three proofs are used before the loop induction in the main
;;; proof.

;; Proof fix_constant_arrays fixes problems with the initialization of
;; the large constant arrays WORDS_FOR_DATA_STRUC_TYPE and CMDS_FOR_DATA_STRUC_TYPE.

(defproof fix_constant_arrays
    "(setflag traceflag on,
      comment \ "Beginning interpretation of fix_constant_arrays... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      ; Go to declaration of WORDS_FOR_DATA_STRUC_TYPE
      go
         #words_for_data_struc_type'first = 1,
      apply 3,

      ; Fix declaration of WORDS_FOR_DATA_STRUC_TYPE
      eval (load \ "/u/versys/msx/array_constant \ "),
      apply,
      eval (load \ "/u/versys/msx/correct_array_constant \ "),

      ; Go to declaration of CMDS_FOR_DATA_STRUC_TYPE
      go
         #cmds_for_data_struc_type'first = 1,
      apply 3,

      ; Fix declaration of CMDS_FOR_DATA_STRUC_TYPE
      eval (load \ "/u/versys/msx/array_constant \ "),
      apply,
      eval (load \ "/u/versys/msx/correct_array_constant \ "),

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment \ "Finished with interpretation of fix_constant_arrays... \ ",
      setflag traceflag off)")

;; Proof go_to_beginning_of_loop symbolically executes until the beginning of the
;; main infinite loop in the program.

(defproof go_to_beginning_of_loop
    "(setflag traceflag on,
      comment \ "Beginning interpretation of go_to_beginning_of_loop... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      go #msx_program_final_version \\ pc = at(loop_begin),

      interpret printdate_if_wanted,
      setflag traceflag on,
      comment  \ "Finished with interpretation of go_to_beginning_of_loop... \ ",
      setflag traceflag off)")

;; Proof do_lets renames various things so that they can be easily referred to
;; later (e.g., in quantifier commands).

(defproof do_lets
    "(setflag traceflag on,
      comment  \ "Beginning interpretation of do_lets... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      letsd loopsd = u(1),
      let old_universe = .msx_program_final_version,
      letq input_condition = q(1),
      letq allowed_message_ids = q(2),
      letq msg_output_begins_at_definition = q(3),
      letq msg_input_begins_at_definition = q(4),
      letq msg_out_lh_definition = q(5),
      letq msg_in_lh_definition = q(6),
      letq checksum_definition = q(7),
      letq checksum_first_k_bytes_message_n_definition = q(8),

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of do_lets... \ ",
      setflag traceflag off)")

;;; The remaining proofs are used in the loop induction in the main proof.
;;; These are split into those used in the base case, and those used in the step
;;; case.


;; Proof base_case.proof proves the base case of the loop induction in the main proof

(defproof base_case.proof
    "(setflag traceflag on,
      comment  \ "Beginning interpretation of base_case.proof... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      ; All goals are vacuously true except those concerning the equality of the
      ; the standard input counter and msg_input_begins_at(1), and of the
      ; standard output counter and msg_output_begins_at(1)

      provebyinstantiation
                using: msg_input_begins_at_definition
                substitutions: (x=1),
      provebyinstantiation
                using: msg_output_begins_at_definition
                substitutions: (x=1),
      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of base_case.proof... \ ",
      setflag traceflag off,
      close)")


;; Proof instantiate_kth_input instantiates the input condition with the value k.

(defproof instantiate_kth_input
    "(setflag traceflag on,
      comment  \ "Beginning interpretation of instantiate_kth_input... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      provebyinstantiation
                using: input_condition_instantiated_on_n
                substitutions: (z=k),

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of instantiate_kth_input... \ ",
      setflag traceflag off)")

;; Proof produce_input_information instantiates the input condition the appropriate
;; number of times so that the prover knows about the inputs for the nth message.

(defproof produce_input_information
    "(setflag traceflag on,
      comment  \ "Beginning interpretation of produce_input_information... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      provebyinstantiation
                using: input_condition
                substitutions: (x=n),
      letq input_condition_instantiated_on_n = q(1),
      repeat instantiate_kth_input iterating on k from 1 to msg_in_lh(n),

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of produce_input_information... \ ",
      setflag traceflag off)")

;; Proof go_to_return_from_call_to_ret_cmd_buf_and_status symbolically executes
;; until the exit from the procedure RET_CMD_BUF_AND_STATUS.

(defproof go_to_return_from_call_to_ret_cmd_buf_and_status
    "(setflag traceflag on,
       comment  \ "Beginning interpretation of
                  go_to_return_from_call_to_ret_cmd_buf_and_status... \ ",
       setflag traceflag off,

       interpret printdate_if_wanted,

      go
       #msx_program_final_version \\ pc =
         exited(msx_program_final_version.serial_dig_cmds.ret_cmd_buf_and_status),

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of
                  go_to_return_from_call_to_ret_cmd_buf_and_status... \ ",
      setflag traceflag off)")

;;; The following proofs are needed as SDVS does not handle array assignments in the
;;; best possible way. This will eventually be fixed.


;; Proof consider_cmd_status_buf_element considers one element of the data structure
;; CMD_STATUS_BUF in SERIAL_DIG_CMDS.

(defproof consider_cmd_status_buf_element
    "(setflag traceflag on,
       comment  \ "Beginning interpretation of consider_cmd_status_buf_element... \ ",
       setflag traceflag off,

       interpret printdate_if_wanted,

       consider .cmd_status_buf[k],

       interpret printdate_if_wanted,

       setflag traceflag on,
       comment  \ "Finished with interpretation of consider_cmd_status_buf_element... \ ",
       setflag traceflag off)")

;; Proof notice_cmd_bufs_records_equal_implies_fields_equal forces the prover to notice
;; that if the kth elements of the CMD_BUF data structures in SERIAL_DIG_CMDS and
;; APP_MSGS are equal, their fields are equal.

(defproof notice_cmd_bufs_records_equal_implies_fields_equal
    "(setflag traceflag on,
       comment  \ "Beginning interpretation of
                      notice_cmd_bufs_records_equal_implies_fields_equal... \ ",
       setflag traceflag off,

       interpret printdate_if_wanted,

      notice
        .app_msgs.cmd_buf[k] = .cmd_buf[k]    -->
         .record(app_msgs.cmd_buf[k],first_byte)  = .record(cmd_buf[k],first_byte),
      notice
```

```
        .app_msgs.cmd_buf[k] = .cmd_buf[k]    -->
         .record(app_msgs.cmd_buf[k],second_byte) = .record(cmd_buf[k],second_byte),
      notice
        .app_msgs.cmd_buf[k] = .cmd_buf[k]    -->
         .record(app_msgs.cmd_buf[k],third_byte)  = .record(cmd_buf[k],third_byte),
      notice
        .app_msgs.cmd_buf[k] = .cmd_buf[k]    -->
         .record(app_msgs.cmd_buf[k],fourth_byte) = .record(cmd_buf[k],fourth_byte),

       interpret printdate_if_wanted,

       setflag traceflag on,
       comment  \ "Finished with interpretation of
                     notice_cmd_bufs_records_equal_implies_fields_equal... \ ",
       setflag traceflag off)")

;; Proof consider_app_msgs.cmd_status_buf_element considers one element of the
;; data structure CMD_STATUS_BUF in APP_MSGS.

(defproof consider_app_msgs.cmd_status_buf_element
    "(setflag traceflag on,
       comment  \ "Beginning interpretation of
                     consider_app_msgs.cmd_status_buf_element... \ ",
       setflag traceflag off,

       interpret printdate_if_wanted,

       consider .app_msgs.cmd_status_buf[k],

       interpret printdate_if_wanted,

       setflag traceflag on,
       comment  \ "Finished with interpretation of
                     consider_app_msgs.cmd_status_buf_element... \ ",
       setflag traceflag off)")

;; Proof fix_array_values_after_call_to_ret_cmd_buf_and_status uses the
;; three proofs above to "remind" the prover of certain information about
;; the arrays passed out of RET_CMD_BUF_AND_STATUS.

(defproof fix_array_values_after_call_to_ret_cmd_buf_and_status
    "(setflag traceflag on,
       comment  \ "Beginning interpretation of
                fix_array_values_after_call_to_ret_cmd_buf_and_status... \ ",
       setflag traceflag off,

       interpret printdate_if_wanted,

       apply u(1),

       repeat consider_cmd_status_buf_element
         iterating on k from 1 to msg_in_lh(n)/4 + 1,

       repeat notice_cmd_bufs_records_equal_implies_fields_equal
         iterating on k from 1 to msg_in_lh(n)/4 + 1,
      repeat consider_app_msgs.cmd_status_buf_element
        iterating on k from 1 to msg_in_lh(n)/4 + 1,

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of
                 fix_array_values_after_call_to_ret_cmd_buf_and_status... \ ",
      setflag traceflag off)")

;;; The following section contains the adalemmas for the intrinsic functions
;;; used in the program and proofs which use them. Note that the postconditions
;;; of the adalemmas are as weak as possible, i.e., geared for input values which
;;; actually occur in the program. These lemmas are not proved as they replace calls
;;; to 1750A code.


;; Adalemma or_word.adalemma is the adalemma for the OR_WORD intrinsic function.

(defadalemma or_word.adalemma
    "/u/versys/msx/msx_program_final_version.a"
     or_word msx_program_final_version.intrinsic_functions.or_word

   ("all.high(.or_word.x) & all.low(.or_word.y)")   (or_word)

   ("(#or_word = .or_word.x + .or_word.y & is.word(#or_word))"))


;; Proof handle_first_or_word_call simply invokes or_word.adalemma and is used
;; only on the first call to OR_WORD.

(defproof handle_first_or_word_call
    "(setflag traceflag on,
      comment  \ "Beginning interpretation of handle_first_or_word_call... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      invokeadalemma or_word.adalemma,

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of handle_first_or_word_call... \ ",
      setflag traceflag off)")

;; Lemma all.low.times.256.is.all.high states that if x is all.low, then 256*x
;; is all.high.

(deflemma all.low.times.256.is.all.high
   "x ge 0 & x le 255
      --> (256 * x ge 0 & 256 * x le 65535) &
             (256 * x) mod 256 = 0"
   (x) nil nil (ok) ;
   :proof "(provelemma all.low.times.256.is.all.high
             proof:
               (read  \ "axioms/mod.axioms \ ",
                provebyaxiom 0 mod 256 = (0 + x * 256) mod 256
                  using: modmult))")

;; Proof handle_remaining_or_word_calls handles all calls to OR_WORD except the first.
;; It uses the lemma all.low.times.256.is.all.high to ensure the precondition of
;; or_word.adalemma is met before it is invoked.

(defproof handle_remaining_or_word_calls
    "(setflag traceflag on,
      comment  \ "Beginning interpretation of handle_remaining_or_word_calls... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      provebylemma
        (256 * .stdin[(((8*(.word_cntr -3))+3)/3)
             + msg_input_begins_at(n) ] ge 0 &
         256 * .stdin[(((8*(.word_cntr -3))+3)/3)
             + msg_input_begins_at(n) ] le 65535) &
        (256 * .stdin[(((8*(.word_cntr -3))+3)/3)
             + msg_input_begins_at(n) ]) mod 256 = 0
               using: all.low.times.256.is.all.high,

      invokeadalemma or_word.adalemma,

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment
       \ "Finished with interpretation of handle_remaining_or_word_calls... \ ",
      setflag traceflag off)")

;; Proof handle_or_word_call uses the proof handle_first_or_word_call on the
;; first call to OR_WORD, handle_remaining_or_word_calls on subsequent calls.
;; This is done because the lemma all.low.times.256.is.all.high is not needed
;; on the first call since the first argument is then 2, and the prover knows
;; all.high(256*2) without any help.

(defproof handle_or_word_call
    "(setflag traceflag on,
      comment  \ "Symbolic execution is at the entry
                  to the intrinsic function OR_WORD \ ",
      comment  \ "This function will be handled by handle_or_word_call, \ ",
      comment  \ "which invokes an adalemma \ ",
      comment  \ "Beginning interpretation of handle_or_word_call... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      if not .build_in_progress
         then handle_first_or_word_call,
      if .build_in_progress
         then handle_remaining_or_word_calls,

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of handle_or_word_call... \ ",
      setflag traceflag off)")

;; Adalemma and_word.lemma is the adalemma for the AND_WORD intrinsic function.

(defadalemma and_word.adalemma
    "/u/versys/msx/msx_program_final_version.a"

    and_word msx_program_final_version.intrinsic_functions.and_word
   ("true")   (and_word)

   ("(.and_word.y = 65280 --> #and_word = (.and_word.x/256)*256) &
     (.and_word.y = 255 --> #and_word = .and_word.x mod 256) &
     (.and_word.x = 65280 --> #and_word = (.and_word.y/256)*256) &
     (.and_word.x = 255 --> #and_word = .and_word.y mod 256) &
     is.word(#and_word)"))

;; Proof handle_and_word_call simply invokes the adalemma and_word.adalemma.

(defproof handle_and_word_call
    "(setflag traceflag on,
      comment  \ "Symbolic execution is at the entry
                    to the intrinsic function AND_WORD \ ",
      comment  \ "This function will be handled by handle_and_word_call, \ ",
      comment  \ "which invokes an adalemma \ ",
      comment  \ "Beginning interpretation of handle_and_word_call... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      invokeadalemma and_word.adalemma,

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of handle_and_word_call... \ ",
      setflag traceflag off)")

;; Adalemma and_i is the adalemma for the AND_I intrinsic function.

(defadalemma and_i.adalemma
    "/u/versys/msx/msx_program_final_version.a"

     and_i msx_program_final_version.intrinsic_functions.and_i
   ("true")   (and_i)

   ("(.and_i.y = 7 --> #and_i = .and_i.x mod 8) &  is.word(#and_i)"))
```

```
;; Proof handle_and_i_call simply invokes the adalemma and_i.adalemma.

(defproof handle_and_i_call
    "(setflag traceflag on,
      comment  \ "Symbolic execution is at the entry
                   to the intrinsic function AND_I \ ",
      comment  \ "This function will be handled by handle_and_i_call, \ ",
      comment  \ "which invokes an adalemma \ ",
      comment  \ "Beginning interpretation of handle_and_i_call... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      invokeadalemma and_i.adalemma,

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of handle_and_i_call... \ ",
      setflag traceflag off)")

;; Adalemma and_byte is the adalemma for the AND_BYTE intrinsic function.

(defadalemma and_byte.adalemma
    "/u/versys/msx/msx_program_final_version.a"

    and_byte msx_program_final_version.intrinsic_functions.and_byte
   ("true")   (and_byte)

   ("(.and_byte.y = 127 --> #and_byte = .and_byte.x mod 128) & is.word(#and_byte)"))



;; Proof handle_and_byte_call simply invokes the adalemma and_byte.adalemma.

(defproof handle_and_byte_call
    "(setflag traceflag on,
      comment  \ "Symbolic execution is at the entry
                   to the intrinsic function AND_BYTE \ ",
      comment  \ "This function will be handled by handle_and_byte_call, \ ",
      comment  \ "which invokes an adalemma \ ",
      comment  \ "Beginning interpretation of handle_and_byte_call... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      invokeadalemma and_byte.adalemma,

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of handle_and_byte_call... \ ",
      setflag traceflag off)")

;; Adalemma shift_logical_word.adalemma is the adalemma for the SHIFT_LOGICAL_WORD
;; intrinsic function.

(defadalemma shift_logical_word.adalemma
    "/u/versys/msx/msx_program_final_version.a"

    shift_logical_word msx_program_final_version.intrinsic_functions.shift_logical_word
    ("true")   (shift_logical_word)

    ("(.shift_logical_word.y = -8 -->
         (#shift_logical_word =
            .shift_logical_word.x / 256 & is.byte(#shift_logical_word))) &
      (is.byte(.shift_logical_word.x) & .shift_logical_word.y = 8 -->
         (#shift_logical_word =
            .shift_logical_word.x * 256 & is.word(#shift_logical_word)))"))



;; Proof handle_shift_logical_word_call simply invokes the adalemma
;; shift_logical_word.adalemma.


(defproof handle_shift_logical_word_call
    "(setflag traceflag on,
      comment  \ "Symbolic execution is at the entry
                    to the intrinsic function SHIFT_LOGICAL_WORD \ ",
      comment  \ "This function will be handled by handle_shift_logical_word_call, \ ",
      comment  \ "which invokes an adalemma \ ",
      comment  \ "Beginning interpretation of handle_shift_logical_word_call... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      invokeadalemma shift_logical_word.adalemma,

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment
        \ "Finished with interpretation of handle_shift_logical_word_call... \ ",
      setflag traceflag off)")

;; Adalemma xor_word.adalemma is the adalemma for the XOR_WORD intrinsic function.

(defadalemma xor_word.adalemma
    "/u/versys/msx/msx_program_final_version.a"

xor_word msx_program_final_version.intrinsic_functions.xor_word
   ("true")   (xor_word)

   ("(#xor_word = sdvs_xor_word(.xor_word.x,.xor_word.y) & is.word(#xor_word))"))

(defproof handle_xor_word_call
    "(setflag traceflag on,
      comment  \ "Symbolic execution is at the entry
                    to the intrinsic function XOR_WORD \ ",
      comment  \ "This function will be handled by handle_xor_word_call, \ ",
      comment  \ "which invokes an adalemma \ ",
      comment  \ "Beginning interpretation of handle_xor_word_call... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      invokeadalemma xor_word.adalemma,

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of handle_xor_word_call... \ ",
      setflag traceflag off)")


;;; The following proofs handle problems with array assignments after the call to
;;; MANAGE_MSG_RETRIEVAL.

;; Proof consider_app_msg_out_element considers the kth element of .app_msg_out.

(defproof consider_app_msg_out_element
    "(setflag traceflag on,
      comment  \ "Beginning interpretation of consider_app_msg_out_element... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      consider .app_msg_out[k],

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of consider_app_msg_out_element... \ ",
      setflag traceflag off)")

;; Proof fix_array_values_after_call_to_manage_msg_retrieval "reminds" the prover
;; of information about the array .app_msg_out.

(defproof fix_array_values_after_call_to_manage_msg_retrieval
    "(setflag traceflag on,
      comment  \ "Symbolic execution is at the exit from the
                MANAGE_MSG_RETRIEVAL procedure \ ",
      comment  \ "Beginning interpretation of
                fix_array_values_after_call_to_manage_msg_retrieval... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      apply u(1),

      repeat consider_app_msg_out_element
        iterating on k from 1 to (1 + msg_out_lh(n)),

      interpret printdate_if_wanted,
      setflag traceflag on,
      comment  \ "Finished with interpretation of
                 fix_array_values_after_call_to_manage_msg_retrieval... \ ",
      setflag traceflag off)")


;; Lemma checksum.lemma is used in proof prove_checksum_correct.

(deflemma checksum.lemma
   "x ge 0 & y gt 0 --> (x / y) * y + x mod y = x"
   (x y) nil nil nil
   :proof "(provelemma checksum.lemma
              proof:
                (read \"axioms/rem.axioms\",
                 read \"axioms/mod.axioms\",
                 mcases

                   (case: x gt 0

                     proof:
                       (provebyaxiom x mod y = x rem y
                          using: modrem1,
                        provebyaxiom x = (x / y) * y + x rem y
                          using: remdef)

                    case: x = 0

                     proof:
                       (read \"axioms/div.axioms\",
                        provebyaxiom x / y = 0
                          using: diveq0,
                        provebyaxiom x mod y = x rem y
                          using: modrem1,
                        provebyaxiom x = (x / y) * y + x rem y
                          using: remdef))))")

;; Proof instantiate_checksum_message_n instantiates k for n in the quantified
;; formula checksum_message_n .

(defproof instantiate_checksum_message_n
    "(setflag traceflag on,
      comment  \ "Beginning interpretation of instantiate_checksum_message_n ... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      provebyinstantiation
        using: checksum_message_n
        substitutions: (k=z),

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of instantiate_checksum_message_n \ ",
      setflag traceflag off)")

;; Proof prove_checksum_correct proves that the result of the checksum
;; calculation has the same value as the output word formed from the
;; input checksum bytes.


(defproof prove_checksum_correct
    "( setflag traceflag on,
      comment \"Beginning interpretation of prove_checksum_correct...\",
      setflag traceflag off,

      interpret printdate_if_wanted,

      provebyinstantiation
        using: checksum_definition
        substitutions: (x=n),

      provebylemma (checksum(n) / 256) *  256 +  checksum(n) mod 256
                   = checksum(n)
        using: checksum.lemma,

      provebyinstantiation
        using: checksum_first_k_bytes_message_n_definition
        substitutions: (n=n),

      letq checksum_message_n = q(1),

      repeat instantiate_checksum_message_n
        iterating on z from 0 to msg_out_lh(n) - 1,

      apply,

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment \"Finished with interpretation of prove_checksum_correct...\",
      setflag traceflag off)")

;; Proof loop_body symbolically executes until it reaches the call of an
;; intrinsic function, the exit from MANAGE_MSG_RETRIEVAL, the proof label
;; after_checksum calculation, or the top of the loop. It uses the
;; appropriate proof (prefixed with handle_ ) on the call of an intrinsic function,
;; fixes array values after_MSG_RETRIEVAL with the proof
;; fix_array_values_after_call_to_manage_msg_retrieval, and uses
;; prove_checksum_correct after the checksum calculation to prove the checksum
;; values are correct.


(defproof loop_body
    "(setflag traceflag on,
      comment  \ "Beginning interpretation of loop_body... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,
go
   ;Call to an intrinsic function

   (#msx_program_final_version \\ pc =
      at(msx_program_final_version.intrinsic_functions.or_word)) or
   (#msx_program_final_version \\ pc =
      at(msx_program_final_version.intrinsic_functions.xor_word)) or
   (#msx_program_final_version \\ pc =
      at(msx_program_final_version.intrinsic_functions.and_i)) or
   (#msx_program_final_version \\ pc =
      at(msx_program_final_version.intrinsic_functions.and_word)) or
   (#msx_program_final_version \\ pc =
      at(msx_program_final_version.intrinsic_functions.and_byte)) or
   (#msx_program_final_version \\ pc =
      at(msx_program_final_version.intrinsic_functions.shift_logical_word)) or

   ;After MANAGE_MSG_RETRIEVAL

   (#msx_program_final_version \\ pc =
      exited(msx_program_final_version.app_msgs.manage_msg_retrieval)) or

   ;After checksum calculation

   (#msx_program_final_version \\ pc =
      at(after_checksum_calculation)) or

   ;Beginning of loop

   formula(loopsd),

   ; Handle intrinsic calls with adalemmas

   if .msx_program_final_version \\ pc =
      at(msx_program_final_version.intrinsic_functions.or_word) then
       handle_or_word_call,
   if .msx_program_final_version \\ pc =
      at(msx_program_final_version.intrinsic_functions.xor_word) then
       handle_xor_word_call,
   if .msx_program_final_version \\ pc =
      at(msx_program_final_version.intrinsic_functions.and_word) then
       handle_and_word_call,
   if .msx_program_final_version \\ pc =
      at(msx_program_final_version.intrinsic_functions.and_i) then
       handle_and_i_call,
   if .msx_program_final_version \\ pc =
      at(msx_program_final_version.intrinsic_functions.and_byte) then
       handle_and_byte_call,
   if .msx_program_final_version \\ pc =
      at(msx_program_final_version.intrinsic_functions.shift_logical_word) then
       handle_shift_logical_word_call,

   ;Fix array values

   if .msx_program_final_version \\ pc =
            exited(msx_program_final_version.app_msgs.manage_msg_retrieval) then
             fix_array_values_after_call_to_manage_msg_retrieval,

         ;Prove checksum bytes have correct value

         if .msx_program_final_version \\ pc =
            at(after_checksum_calculation) then
             prove_checksum_correct,

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of loop_body... \ ",
      setflag traceflag off)")

;; Proof handle_intrinsics_until_end_of_program interprets loop_body until the top
;; of the loop.

(defproof handle_intrinsics_until_end_of_program
    "(setflag traceflag on,
      comment  \ "Beginning interpretation of
                handle_intrinsics_until_end_of_program... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      loop until formula(loopsd) with loop_body,

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of
                handle_intrinsics_until_end_of_program... \ ",
      setflag traceflag off)")

;; Lemma queue_bounds.lemma is needed to prove that after the variable Q_HEAD and
;; Q_TAIL are updated, they are still legitimate array bounds (i.e., between one and
;; thirty inclusive). State deltas that correspond to statements which use these
;; variables as array indices have preconditions requiring that these variables
;; are in bounds, so statements that the original values of these variables are in
;; bounds are part of the loop invariant in the main proof.

(deflemma queue_bounds.lemma
   "z ge 1 & z le 30
       --> z mod 30 + 1 ge 1 & z mod 30 + 1 le 30"
   (z) nil nil nil

   :proof "(provelemma queue_bounds.lemma
              proof:
                (read  \ "axioms/mod.axioms \ ",
                 read  \ "axioms/rem.axioms \ ",
                 read  \ "axioms/div.axioms \ ",
                 cases z lt 30
                   then proof:
                     (provebyaxiom z / 30 = 0
                          using: diveq0,
                        provebyaxiom z = (z / 30) * 30 + z rem 30
                          using: remdef,
                        provebyaxiom z mod 30 = z rem 30
                          using: modrem1)
                     else proof: ))")

;; Proof finish_off_non_output_goals proves that all of the goals which the prover
;; does not know to be true after symbolically executing through the loop with the
;; exception of the goal involving the output condition.

(defproof finish_off_non_output_goals
    "(setflag traceflag on,
      comment  \ "Beginning interpretation of finish_off_non_output_goals... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      ; Proves that the statements concerning the queue variables
      ; in the invariant are re-established

      provebylemma old_q_head mod 30 + 1 ge 1 &
                   old_q_head mod 30 + 1 le 30
                    using: queue_bounds.lemma,

      ; Input/output counters have been incremented appropriately

      provebyinstantiation
         using: msg_output_begins_at_definition
         substitutions: (x=n + 1),
      provebyinstantiation
         using: msg_input_begins_at_definition
         substitutions: (x=n + 1),

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of finish_off_non_output_goals... \ ",
      setflag traceflag off)")


;;; The following section contains proof commands used to prove the output condition
;;; in the n = x case. At the end of this case, the prover knows (i.e., can simplify
;;; to true) the output condition at 1, 2, ...., msg_out_lh(x), but doesn't know the
;;; actual output condition. So we are the position of having to convince the prover
;;; of forall z (1 le z & z le msg_out_lh(x) --> P(z)) when it already knows P(1),
;;; P(2), ..., P(msg_out_lh(x) - 1), P(msg_out_lh(x)).


;; Proof notice_output_condition_at_k_quantified_over_z notices forall z P(k)
;; when P(k) is true and puts it on the list of quantifiers (N.B.: z does not
;; occur free in P(k) )

(defproof notice_output_condition_at_k_quantified_over_z
    "(setflag traceflag on,
      comment  \ "Beginning interpretation of
                 notice_output_condition_at_k_quantified_over_z... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      notice
       forall z (.stdout[msg_output_begins_at(x) + (k - 1)]
                   = mk.word(.stdin[msg_input_begins_at(x) + (8 * k - 5) / 3],
                              .stdin[msg_input_begins_at(x) + (8 * k - 1) / 3])),

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of
                 notice_output_condition_at_k_quantified_over_z... \ ",
      setflag traceflag off)")

;; Proof another_version_of_output_condition_at_k_quantified_over_z proves
;; forall z (z=k --> P(k)) from forall z P(k)

(defproof another_version_of_output_condition_at_k_quantified_over_z
    "(setflag traceflag on,
      comment  \ "Beginning interpretation of
                 another_version_of_output_condition_at_k_quantified_over_z... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      provebygeneralization
        forall z (z = k -->
              .stdout[msg_output_begins_at(x) + (z - 1)]
                = mk.word(.stdin[msg_input_begins_at(x) + (8 * z - 5) / 3] ,
                           .stdin[msg_input_begins_at(x) + (8 * z - 1) / 3]))

        using: (forall z (.stdout[msg_output_begins_at(x) + (k - 1)]
                           = mk.word(.stdin[msg_input_begins_at(x) + (8 * k - 5) / 3] ,
                                      .stdin[msg_input_begins_at(x) + (8 * k - 1) / 3]))),

      interpret printdate_if_wanted,

      setflag traceflag on,
      comment  \ "Finished with interpretation of
                 another_version_of_output_condition_at_k_quantified_over_z... \ ",
      setflag traceflag off)")

;; Proof prove_output_condition_for_z_equal_to_1 proves
;; forall z (1 le z & z lt 2 --> P(z)) from forall z (z = 1 --> P(z))

(defproof prove_output_condition_for_z_equal_to_1
    "(setflag traceflag on,
      comment  \ "Beginning interpretation of
                     prove_output_condition_for_z_equal_to_1... \ ",
      setflag traceflag off,
     interpret printdate_if_wanted,

     provebymakeboundedquantifier
       forall z (1  le z & z  lt 2  -->
              .stdout[msg_output_begins_at(x) + (z - 1)]
                 = mk.word(.stdin[msg_input_begins_at(x) + (8 * z  - 5) / 3],
                          .stdin[msg_input_begins_at(x) + (8 * z  - 1) / 3]))

       using: (forall z (z = 1  -->
              .stdout[msg_output_begins_at(x) + (z - 1)]
                = mk.word(.stdin[msg_input_begins_at(x) + (8 * z - 5)  / 3],
                          .stdin[msg_input_begins_at(x) + (8 * z - 1)  / 3]))),

     interpret printdate_if_wanted,

     setflag traceflag on,
     comment  \ "Finished with interpretation of
                  prove_output_condition_for_z_equal_to_1... \ ",
     setflag traceflag off)")

;; Proof prove_output_condition_for_z_ranging_from_1_to_k_plus_1 proves
;; forall z ( 1 le z & z lt k+1 --> P(z)) from forall z (1 le z & z lt k --> P(z))
;; and forall z (z=k -->P(z))

(defproof prove_output_condition_for_z_ranging_from_1_to_k_plus_1
    "(setflag traceflag on,
     comment  \ "Beginning interpretation of
              prove_output_condition_for_z_ranging_from_1_to_k_plus_1... \ ",
     setflag traceflag off,

     interpret printdate_if_wanted,

     provebymakeboundedquantifier
       forall z (1 le z & z lt k+1 -->
                 .stdout[msg_output_begins_at(x) + (z - 1)]
                   = mk.word(.stdin[msg_input_begins_at(x) + (8 * z - 5) / 3] ,
                             .stdin[msg_input_begins_at(x) + (8 * z - 1) / 3]))

        using: (forall z (1 le z & z lt k  -->
                 .stdout[msg_output_begins_at(x) + (z - 1)]
                   = mk.word(.stdin[msg_input_begins_at(x) + (8 * z - 5) / 3] ,
                             .stdin[msg_input_begins_at(x) + (8 * z - 1) / 3])),

                forall z (z = k -->
                 .stdout[msg_output_begins_at(x) + (z - 1)]
                   = mk.word(.stdin[msg_input_begins_at(x) + (8 * z - 5) / 3] ,
                             .stdin[msg_input_begins_at(x) + (8 * z - 1) / 3]))),

     interpret printdate_if_wanted,

     setflag traceflag on,
     comment  \ "Finished with interpretation of
              prove_output_condition_for_z_ranging_from_1_to_k_plus_1... \ ",
     setflag traceflag off)")
```

```
;; Proof prove_actual_output_condition is the last step in the proof
;; finish_off_output_goal and proves the actual output condition.

(defproof prove_actual_output_condition
    "(setflag traceflag on,
      comment  \ "Beginning interpretation of prove_actual_output_condition... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      provebymakeboundedquantifier
       forall z ((1 le z & z le msg_out_lh(x)) -->
                  (.stdout[msg_output_begins_at(x) + (z - 1)]
                     = mk.word(.stdin[msg_input_begins_at(x) + (8 * z - 5) / 3] ,
                               .stdin[msg_input_begins_at(x) + (8 * z - 1) / 3])))

       using: (forall z ((1 le z & z lt k) -->
                  (.stdout[msg_output_begins_at(x) + (z - 1)]
                     = mk.word(.stdin[msg_input_begins_at(x) + (8 * z - 5) / 3] ,
                               .stdin[msg_input_begins_at(x) + (8 * z - 1) / 3]))),

               forall z ((z = k)  -->
                  (.stdout[msg_output_begins_at(x) + (z - 1)]
                     = mk.word(.stdin[msg_input_begins_at(x) + (8 * z - 5) / 3] ,
                               .stdin[msg_input_begins_at(x) + (8 * z - 1) / 3])))),

     interpret printdate_if_wanted,

     setflag traceflag on,
     comment  \ "Finished with interpretation of prove_actual_output_condition... \ ",
     setflag traceflag off)")

;; Proof finish_off_output_goal uses the above proof commands to prove the
;; output condition.

(defproof finish_off_output_goal
    "(setflag traceflag on,
      comment  \ "Beginning interpretation of finish_off_output_goal... \ ",
      setflag traceflag off,

      interpret printdate_if_wanted,

      repeat notice_output_condition_at_k_quantified_over_z
               iterating on k from 1 to msg_out_lh(x),

      eval (load  \ "/u/versys/msx/dotted_subexpressions \ "),

      repeat another_version_of_output_condition_at_k_quantified_over_z
               iterating on k from 1 to msg_out_lh(x),

      eval (load  \ "/u/versys/msx/correct_dotted_subexpressions \ "),

      interpret prove_output_condition_for_z_equal_to_1,

      repeat prove_output_condition_for_z_ranging_from_1_to_k_plus_1
```

```
          iteration on k from 2 to msg_out_lh(x) - 1,

   repeat prove_actual_output_condition
          iterating on k from msg_out_lh(x) to msg_out_lh(x),

   interpret printdate_if_wanted,

   setflag traceflag on,
   comment  \ "Finished with interpretation of finish_off_output_goal... \ ",
   setflag traceflag off)")



;; Proof every_case.proof is the generic proof for every case in the step case
;; of the induction.

(defproof every_case.proof
    "(setflag traceflag on,
   comment  \ "Beginning interpretation of every_case.proof... \ ",
   setflag traceflag off,

   interpret printdate_if_wanted,

   ; Want to name current value of q_head to use in queue_bounds.lemma

   let old_q_head = .q_head,

   ; These instantiations are needed so that the prover knows the concrete
   ; values of the input and output length of the current message, as
   ; these are used in repeat loops.

   provebyinstantiation
         using: msg_in_lh_definition
         substitutions: (x = n),
   provebyinstantiation
         using: msg_out_lh_definition
         substitutions: (x = n),

   mcases

      (case: n lt x
           proof:
            (setflag traceflag on,
             comment  \ "Beginning interpretation of n lt x case... \ ",
             setflag traceflag off,

             interpret printdate_if_wanted,

             ; Needed to speed up proof (see comments at definitions)
             interpret separate_input_places,
             interpret separate_output_places,

             ; Common part of both cases

             interpret produce_input_information,
          interpret go_to_return_from_call_to_ret_cmd_buf_and_status,
          interpret fix_array_values_after_call_to_ret_cmd_buf_and_status,
          interpret handle_intrinsics_until_end_of_program,
          interpret finish_off_non_output_goals,

          ; End of common part

          comment  \ "Attempting to close on goals for n lt x case... \ ",
          interpret printdate_if_wanted,
          close)

    case: n = x
         proof:
         (setflag traceflag on,
          comment  \ "Beginning interpretation of n = x case... \ ",
          setflag traceflag off,

          interpret printdate_if_wanted,

          ; Common part of both cases

          interpret produce_input_information,
          interpret go_to_return_from_call_to_ret_cmd_buf_and_status,
          interpret fix_array_values_after_call_to_ret_cmd_buf_and_status,
          interpret handle_intrinsics_until_end_of_program,
          interpret finish_off_non_output_goals,

          ; End of common part

          interpret finish_off_output_goal,
          comment  \ "Attempting to close on goals for n = x case... \ ",
          interpret printdate_if_wanted,
          close)),

   interpret printdate_if_wanted,

   setflag traceflag on,
   comment  \ "Finished with interpretation of every_case.proof... \ ",
   setflag traceflag off)")
```

9.1.7 The versions of step_case.proof used in the main proof

```
;; Proof step_case.proof just instantiates allowed_message_ids with n
;; for x, and then runs every_case.proof on all possible message ids
;; in a multiple cases command. Due to concerns about system crashes,
;; we ran the proof with different versions of step_case.proof. In
;; each version, we proved one or more cases and deferred the rest.
;; All of the versions of step_case.proof used are given (in order)
;; below. The cases handled in each version are given in the second
;; comment line.
```

```
(defproof step_case.proof
    "(setflag traceflag on,
      comment \"Beginning interpretation of step_case.proof...\",
      comment \"This version of step_case.proof handles message ids: 1\",
      setflag traceflag off,

      interpret printdate_if_wanted,


      ; Get the possible message identifiers for n

      provebyinstantiation
                     using: allowed_message_ids
                     substitutions: (x=n),

      ; Run the cases for the message identifiers mentioned in the
      ; comment above, and defer all other cases

      mcases
         (case: msg_id(n) = 1
           proof:
                 (setflag traceflag on,
                  comment \"Currently handling message id 1...\",
                  setflag traceflag off,

                  interpret every_case.proof,

                  setflag traceflag on,
                  comment \"Finished with message id 1....\",
                  setflag traceflag off)
          case: msg_id(n) ~= 1
           proof:
                 (setflag traceflag on,
                  comment \"Deferring proofs for all message ids except : 1 \",
                  setflag traceflag off,
                  defer)),

      interpret printdate_if_wanted,

      close,

      setflag traceflag on,
      comment \"Finished with interpretation of step_case.proof...\",
      setflag traceflag off)")

(defproof step_case.proof
    "(setflag traceflag on,
      comment \"Beginning interpretation of step_case.proof...\",
      comment \"This version of step_case.proof handles message ids: 2 and 3\",
      setflag traceflag off,

      interpret printdate_if_wanted,


      ; Get all the possible message ids for n
    provebyinstantiation
                   using: allowed_message_ids
                   substitutions: (x=n),

    ; Run the cases for the message identifiers mentioned in the
    ; comment above, and defer all other cases

    mcases
       (case: msg_id(n) = 2
         proof:
               (setflag traceflag on,
                comment \"Currently handling message id 2...\",
                setflag traceflag off,

                interpret every_case.proof,

                setflag traceflag on,
                comment \"Finished with message id 2....\",
                setflag traceflag off)

        case: msg_id(n) = 3
         proof:
               (setflag traceflag on,
                comment \"Currently handling message id 3...\",
                setflag traceflag off,

                interpret every_case.proof,

                setflag traceflag on,
                comment \"Finished with message id 3....\",
                setflag traceflag off)


        case: (msg_id(n) ~= 2 & msg_id(n) ~= 3)
         proof:
               (setflag traceflag on,
                comment \"Deferring proofs for all ids except : 2 and 3 ... \",
                setflag traceflag off,

                defer)),

    interpret printdate_if_wanted,

    close,

    setflag traceflag on,
    comment \"Finished with interpretation of step_case.proof...\",
    setflag traceflag off)")


(defproof step_case.proof
   "(setflag traceflag on,
    comment \"Beginning interpretation of step_case.proof...\",
    comment \"This version of step_case.proof handles message ids: 4 and 5\",
setflag traceflag off,

interpret printdate_if_wanted,


; Get all the possible message ids for n

provebyinstantiation
                using: allowed_message_ids
                substitutions: (x=n),

; Run the cases for the message identifiers mentioned in the
; comment above, and defer all other cases

mcases
   (case: msg_id(n) = 4
     proof:
            (setflag traceflag on,
             comment \"Currently handling message id 4...\",
             setflag traceflag off,

             interpret every_case.proof,

             setflag traceflag on,
             comment \"Finished with message id 4....\",
             setflag traceflag off)

    case: msg_id(n) = 5
     proof:
            (setflag traceflag on,
             comment \"Currently handling message id 5...\",
             setflag traceflag off,

             interpret every_case.proof,

             setflag traceflag on,
             comment \"Finished with message id 5....\",
             setflag traceflag off)


    case: (msg_id(n) ~= 4 & msg_id(n) ~= 5)
     proof:
            (setflag traceflag on,
             comment \"Deferring proofs for all ids except : 4 and 5 ... \",
             setflag traceflag off,

             defer)),

interpret printdate_if_wanted,

close,

setflag traceflag on,
comment \"Finished with interpretation of step_case.proof...\",
setflag traceflag off)")
```

```
(defproof step_case.proof
    "(setflag traceflag on,
      comment \"Beginning interpretation of step_case.proof...\",
      comment \"This version of step_case.proof handles message ids: 6 and 7\",
      setflag traceflag off,

      interpret printdate_if_wanted,


      ; Get all the possible message ids for n

      provebyinstantiation
                      using: allowed_message_ids
                      substitutions: (x=n),

      ; Run the cases for the message identifiers mentioned in the
      ; comment above, and defer all other cases

      mcases
         (case: msg_id(n) = 6
          proof:
                 (setflag traceflag on,
                  comment \"Currently handling message id 6...\",
                  setflag traceflag off,

                  interpret every_case.proof,

                  setflag traceflag on,
                  comment \"Finished with message id 6....\",
                  setflag traceflag off)

          case: msg_id(n) = 7
          proof:
                 (setflag traceflag on,
                  comment \"Currently handling message id 7...\",
                  setflag traceflag off,

                  interpret every_case.proof,

                  setflag traceflag on,
                  comment \"Finished with message id 7....\",
                  setflag traceflag off)


          case: (msg_id(n) ~= 6 & msg_id(n) ~= 7)
          proof:
                 (setflag traceflag on,
                  comment \"Deferring proofs for all ids except : 6 and 7 ... \",
                  setflag traceflag off,

                  defer)),

      interpret printdate_if_wanted,
      close,

      setflag traceflag on,
      comment \"Finished with interpretation of step_case.proof...\",
      setflag traceflag off)")

(defproof step_case.proof
    "(setflag traceflag on,
      comment \"Beginning interpretation of step_case.proof...\",
      comment \"This version of step_case.proof handles message ids:
                  8, 9, 10, 11, and 12\",
      setflag traceflag off,

      interpret printdate_if_wanted,


      ; Get all the possible message ids for n

      provebyinstantiation
                      using: allowed_message_ids
                      substitutions: (x=n),

      ; Run the cases for the message identifiers mentioned in the
      ; comment above, and defer all other cases

      mcases
         (case: msg_id(n) = 8
           proof:
                  (setflag traceflag on,
                   comment \"Currently handling message id 8...\",
                   setflag traceflag off,

                   interpret every_case.proof,

                   setflag traceflag on,
                   comment \"Finished with message id 8....\",
                   setflag traceflag off)

          case: msg_id(n) = 9
           proof:
                  (setflag traceflag on,
                   comment \"Currently handling message id 9...\",
                   setflag traceflag off,

                   interpret every_case.proof,

                   setflag traceflag on,
                   comment \"Finished with message id 9....\",
                   setflag traceflag off)

          case: msg_id(n) = 10
           proof:
                  (setflag traceflag on,
                   comment \"Currently handling message id 10...\",
                   setflag traceflag off,
                 interpret every_case.proof,

                 setflag traceflag on,
                 comment \"Finished with message id 10....\",
                 setflag traceflag off)

         case: msg_id(n) = 11
          proof:
                (setflag traceflag on,
                 comment \"Currently handling message id 11...\",
                 setflag traceflag off,

                 interpret every_case.proof,

                 setflag traceflag on,
                 comment \"Finished with message id 11....\",
                 setflag traceflag off)

         case: msg_id(n) = 12
          proof:
                (setflag traceflag on,
                 comment \"Currently handling message id 12...\",
                 setflag traceflag off,

                 interpret every_case.proof,

                 setflag traceflag on,
                 comment \"Finished with message id 12....\",
                 setflag traceflag off)

         case: (msg_id(n) ~= 8 & msg_id(n) ~= 9 & msg_id(n) ~= 10
                 & msg_id(n) ~= 11 & msg_id(n) ~= 12)
          proof:
                (setflag traceflag on,
                 comment \"Deferring proofs for all ids except :
                              8, 9, 10, 11, and 12 ... \",
                 setflag traceflag off,

                 defer)),

     interpret printdate_if_wanted,

     close,

     setflag traceflag on,
     comment \"Finished with interpretation of step_case.proof...\",
     setflag traceflag off)")

(defproof step_case.proof
   "(setflag traceflag on,
     comment \"Beginning interpretation of step_case.proof...\",
     comment \"This version of step_case.proof handles message ids:
               13, 14, 15, 16, 17, 18, 19, and 20\",
     setflag traceflag off,
interpret printdate_if_wanted,


; Get all the possible message ids for n

provebyinstantiation
                using: allowed_message_ids
                substitutions: (x=n),

; Run the cases for the message identifiers mentioned in the
; comment above, and defer all other cases

mcases
   (case: msg_id(n) = 13
     proof:
           (setflag traceflag on,
            comment \"Currently handling message id 13...\",
            setflag traceflag off,

            interpret every_case.proof,

            setflag traceflag on,
            comment \"Finished with message id 13....\",
            setflag traceflag off)

    case: msg_id(n) = 14
     proof:
           (setflag traceflag on,
            comment \"Currently handling message id 14...\",
            setflag traceflag off,

            interpret every_case.proof,

            setflag traceflag on,
            comment \"Finished with message id 14....\",
            setflag traceflag off)

    case: msg_id(n) = 15
     proof:
           (setflag traceflag on,
            comment \"Currently handling message id 15...\",
            setflag traceflag off,

            interpret every_case.proof,

            setflag traceflag on,
            comment \"Finished with message id 15....\",
            setflag traceflag off)

    case: msg_id(n) = 16
     proof:
           (setflag traceflag on,
            comment \"Currently handling message id 16...\",
            setflag traceflag off,
         interpret every_case.proof,

         setflag traceflag on,
         comment \"Finished with message id 16....\",
         setflag traceflag off)

case: msg_id(n) = 17
 proof:
        (setflag traceflag on,
         comment \"Currently handling message id 17...\",
         setflag traceflag off,

         interpret every_case.proof,

         setflag traceflag on,
         comment \"Finished with message id 17....\",
         setflag traceflag off)

case: msg_id(n) = 18
 proof:
        (setflag traceflag on,
         comment \"Currently handling message id 18...\",
         setflag traceflag off,

         interpret every_case.proof,

         setflag traceflag on,
         comment \"Finished with message id 18....\",
         setflag traceflag off)

case: msg_id(n) = 19
 proof:
        (setflag traceflag on,
         comment \"Currently handling message id 19...\",
         setflag traceflag off,

         interpret every_case.proof,

         setflag traceflag on,
         comment \"Finished with message id 19....\",
         setflag traceflag off)

case: msg_id(n) = 20
 proof:
        (setflag traceflag on,
         comment \"Currently handling message id 20...\",
         setflag traceflag off,

         interpret every_case.proof,

         setflag traceflag on,
         comment \"Finished with message id 20....\",
         setflag traceflag off)
        case: (msg_id(n) ~= 13 & msg_id(n) ~= 14 & msg_id(n) ~= 15
               & msg_id(n) ~= 16 & msg_id(n) ~= 17 & msg_id(n) ~= 18
               & msg_id(n) ~= 19 & msg_id(n) ~= 20)
         proof:
               (setflag traceflag on,
                comment \"Deferring proofs for all ids except :
                            13, 14, 15, 16, 17, 18, 19, and 20... \",
                setflag traceflag off,

                defer)),

    interpret printdate_if_wanted,

    close,

    setflag traceflag on,
    comment \"Finished with interpretation of step_case.proof...\",
    setflag traceflag off)")


(defproof step_case.proof
   "(setflag traceflag on,
    comment \"Beginning interpretation of step_case.proof...\",
    comment \"This version of step_case.proof handles message ids:
                21, 22, 23, 24, 25, and 26\",
    setflag traceflag off,

    interpret printdate_if_wanted,


    ; Get the possible message identifiers for n

    provebyinstantiation
                   using: allowed_message_ids
                   substitutions: (x=n),

    ; Run the cases for the message identifiers mentioned in the
    ; comment above, and defer all other cases

    mcases
       (case: msg_id(n) = 21
         proof:
               (setflag traceflag on,
                comment \"Currently handling message id 21...\",
                setflag traceflag off,

                interpret every_case.proof,

                setflag traceflag on,
                comment \"Finished with message id 21....\",
                setflag traceflag off)

        case: msg_id(n) = 22
         proof:
               (setflag traceflag on,
        comment \"Currently handling message id 22...\",
        setflag traceflag off,

        interpret every_case.proof,

        setflag traceflag on,
        comment \"Finished with message id 22....\",
        setflag traceflag off)

case: msg_id(n) = 23
 proof:
       (setflag traceflag on,
        comment \"Currently handling message id 23...\",
        setflag traceflag off,

        interpret every_case.proof,

        setflag traceflag on,
        comment \"Finished with message id 23....\",
        setflag traceflag off)

case: msg_id(n) = 24
 proof:
       (setflag traceflag on,
        comment \"Currently handling message id 24...\",
        setflag traceflag off,

        interpret every_case.proof,

        setflag traceflag on,
        comment \"Finished with message id 24....\",
        setflag traceflag off)

case: msg_id(n) = 25
 proof:
       (setflag traceflag on,
        comment \"Currently handling message id 25...\",
        setflag traceflag off,

        interpret every_case.proof,

        setflag traceflag on,
        comment \"Finished with message id 25....\",
        setflag traceflag off)

case: msg_id(n) = 26
 proof:
       (setflag traceflag on,
        comment \"Currently handling message id 26...\",
        setflag traceflag off,

        interpret every_case.proof,

        setflag traceflag on,
        comment \"Finished with message id 26....\",
              setflag traceflag off)

          case: (msg_id(n) ~= 21 & msg_id(n) ~= 22 & msg_id(n) ~= 23
              & msg_id(n) ~= 24 & msg_id(n) ~= 25 & msg_id(n) ~= 26)
           proof:
              (setflag traceflag on,
               comment \"Deferring proofs for all ids except :
                         21, 22, 23, 24, 25 ,and 26... \",
               setflag traceflag off,

               defer)),

      interpret printdate_if_wanted,

      close,

      setflag traceflag on,
      comment \"Finished with interpretation of step_case.proof...\",
      setflag traceflag off)")

(defproof step_case.proof
    "(setflag traceflag on,
      comment \"Beginning interpretation of step_case.proof...\",
      comment \"This version of step_case.proof handles message ids:
               27, 28, and 29\",
      setflag traceflag off,

      interpret printdate_if_wanted,


      ; Get all the possible message ids for n

      provebyinstantiation
                     using: allowed_message_ids
                     substitutions: (x=n),

      ; Run the cases for the message identifiers mentioned in the
      ; comment above, and defer all other cases

      mcases
         (case: msg_id(n) = 27
           proof:
              (setflag traceflag on,
               comment \"Currently handling message id 27...\",
               setflag traceflag off,

               interpret every_case.proof,

               setflag traceflag on,
               comment \"Finished with message id 27....\",
               setflag traceflag off)

          case: msg_id(n) = 28
           proof:
              (setflag traceflag on,
                comment \"Currently handling message id 28...\",
                setflag traceflag off,

                interpret every_case.proof,

                setflag traceflag on,
                comment \"Finished with message id 28....\",
                setflag traceflag off)

        case: msg_id(n) = 29
         proof:
               (setflag traceflag on,
                comment \"Currently handling message id 29...\",
                setflag traceflag off,

                interpret every_case.proof,

                setflag traceflag on,
                comment \"Finished with message id 29....\",
                setflag traceflag off)

        case: (msg_id(n) ~= 27 & msg_id(n) ~= 28 & msg_id(n) ~= 29)
         proof:
               (setflag traceflag on,
                comment \"Deferring proofs for all ids except :
                            27, 28, and 29 ... \",
                setflag traceflag off,

                defer)),

    interpret printdate_if_wanted,

    close,

    setflag traceflag on,
    comment \"Finished with interpretation of step_case.proof...\",
    setflag traceflag off)")

(defproof step_case.proof
   "(setflag traceflag on,
    comment \"Beginning interpretation of step_case.proof...\",
    comment \"This version of step_case.proof handles message ids: 50\",
    setflag traceflag off,

    interpret printdate_if_wanted,


    ; Get the possible message identifiers for n

    provebyinstantiation
                using: allowed_message_ids
                substitutions: (x=n),

    ; Run the cases for the message identifiers mentioned in the
    ; comment above, and defer all other cases
    mcases
       (case: msg_id(n) = 50
         proof:
               (setflag traceflag on,
                comment \"Currently handling message id 50...\",
                setflag traceflag off,

                interpret every_case.proof,

                setflag traceflag on,
                comment \"Finished with message id 50....\",
                setflag traceflag off)
        case: msg_id(n) ~= 50
         proof:
               (setflag traceflag on,
                comment \"Deferring proofs for all message ids except : 50 \",
                setflag traceflag off,
                defer)),

    interpret printdate_if_wanted,

    close,

    setflag traceflag on,
    comment \"Finished with interpretation of step_case.proof...\",
    setflag traceflag off)")


(defproof step_case.proof
    "(setflag traceflag on,
    comment \"Beginning interpretation of step_case.proof...\",
    comment \"This version of step_case.proof handles message ids: 51 and 52\",
    setflag traceflag off,

    interpret printdate_if_wanted,


    ; Get all the possible message ids for n

    provebyinstantiation
                   using: allowed_message_ids
                   substitutions: (x=n),

    ; Run the cases for the message identifiers mentioned in the
    ; comment above, and defer all other cases

    mcases
       (case: msg_id(n) = 51
         proof:
               (setflag traceflag on,
                comment \"Currently handling message id 51...\",
                setflag traceflag off,

                interpret every_case.proof,
                setflag traceflag on,
                comment \"Finished with message id 51....\",
                setflag traceflag off)

        case: msg_id(n) = 52
         proof:
               (setflag traceflag on,
                comment \"Currently handling message id 52...\",
                setflag traceflag off,

                interpret every_case.proof,

                setflag traceflag on,
                comment \"Finished with message id 52....\",
                setflag traceflag off)


        case: (msg_id(n) ~= 51 & msg_id(n) ~= 52)
         proof:
               (setflag traceflag on,
                comment \"Deferring proofs for all ids except : 51 and 52... \",
                setflag traceflag off,

                defer)),

      interpret printdate_if_wanted,

      close,

      setflag traceflag on,
      comment \"Finished with interpretation of step_case.proof...\",
      setflag traceflag off)")

;;; N.B: We were unable to complete cases 53, 54, 55, and 56 due to
;;; problems with running out of storage, so there are no versions of
;;; step_case.proof which handle these cases.

(defproof step_case.proof
    "(setflag traceflag on,
      comment \"Beginning interpretation of step_case.proof...\",
      comment \"This version of step_case.proof handles message ids: 57\",
      setflag traceflag off,

      interpret printdate_if_wanted,


      ; Get the possible message identifiers for n

      provebyinstantiation
                     using: allowed_message_ids
                     substitutions: (x=n),

      ; Run the cases for the message identifiers mentioned in the
      ; comment above, and defer all other cases
```

```
       mcases
          (case: msg_id(n) = 57
            proof:
                   (setflag traceflag on,
                    comment \"Currently handling message id 57...\",
                    setflag traceflag off,

                    interpret every_case.proof,

                    setflag traceflag on,
                    comment \"Finished with message id 57....\",
                    setflag traceflag off)
           case: msg_id(n) ~= 57
            proof:
                   (setflag traceflag on,
                    comment \"Deferring proofs for all message ids except : 57 \",
                    setflag traceflag off,
                    defer)),

       interpret printdate_if_wanted,

       close,

       setflag traceflag on,
       comment \"Finished with interpretation of step_case.proof...\",
       setflag traceflag off)")

(defproof step_case.proof
    "(setflag traceflag on,
       comment \"Beginning interpretation of step_case.proof...\",
       comment \"This version of step_case.proof handles message ids: 58\",
       setflag traceflag off,

       interpret printdate_if_wanted,


       ; Get the possible message identifiers for n

       provebyinstantiation
                      using: allowed_message_ids
                      substitutions: (x=n),

       ; Run the cases for the message identifiers mentioned in the
       ; comment above, and defer all other cases

       mcases
          (case: msg_id(n) = 58
            proof:
                   (setflag traceflag on,
                    comment \"Currently handling message id 58...\",
                    setflag traceflag off,

                    interpret every_case.proof,
                setflag traceflag on,
                comment \"Finished with message id 58....\",
                setflag traceflag off)
        case: msg_id(n) ~= 58
         proof:
               (setflag traceflag on,
                comment \"Deferring proofs for all message ids except : 58 \",
                setflag traceflag off,
                defer)),

    interpret printdate_if_wanted,

    close,

    setflag traceflag on,
    comment \"Finished with interpretation of step_case.proof...\",
    setflag traceflag off)")

(defproof step_case.proof
  "(setflag traceflag on,
    comment \"Beginning interpretation of step_case.proof...\",
    comment \"This version of step_case.proof handles message ids: 59\",
    setflag traceflag off,

    interpret printdate_if_wanted,


    ; Get the possible message identifiers for n

    provebyinstantiation
                 using: allowed_message_ids
                 substitutions: (x=n),

    ; Run the cases for the message identifiers mentioned in the
    ; comment above, and defer all other cases

    mcases
       (case: msg_id(n) = 59
         proof:
               (setflag traceflag on,
                comment \"Currently handling message id 59...\",
                setflag traceflag off,

                interpret every_case.proof,

                setflag traceflag on,
                comment \"Finished with message id 59....\",
                setflag traceflag off)
        case: msg_id(n) ~= 59
         proof:
               (setflag traceflag on,
                comment \"Deferring proofs for all message ids except : 59 \",
                setflag traceflag off,
                defer)),
        interpret printdate_if_wanted,

        close,

        setflag traceflag on,
        comment \"Finished with interpretation of step_case.proof...\",
        setflag traceflag off)")

(defproof step_case.proof
    "(setflag traceflag on,
        comment \"Beginning interpretation of step_case.proof...\",
        comment \"This version of step_case.proof handles message ids: 60\",
        setflag traceflag off,

        interpret printdate_if_wanted,


        ; Get the possible message identifiers for n

        provebyinstantiation
                        using: allowed_message_ids
                        substitutions: (x=n),

        ; Run the cases for the message identifiers mentioned in the
        ; comment above, and defer all other cases

        mcases
           (case: msg_id(n) = 60
             proof:
                   (setflag traceflag on,
                    comment \"Currently handling message id 60...\",
                    setflag traceflag off,

                    interpret every_case.proof,

                    setflag traceflag on,
                    comment \"Finished with message id 60....\",
                    setflag traceflag off)
            case: msg_id(n) ~= 60
             proof:
                   (setflag traceflag on,
                    comment \"Deferring proofs for all message ids except : 60 \",
                    setflag traceflag off,
                    defer)),

        interpret printdate_if_wanted,

        close,

        setflag traceflag on,
        comment \"Finished with interpretation of step_case.proof...\",
        setflag traceflag off)")

(defproof step_case.proof
    "(setflag traceflag on,
comment \"Beginning interpretation of step_case.proof...\",
comment \"This version of step_case.proof handles message ids: 61 and 62\",
setflag traceflag off,

interpret printdate_if_wanted,


; Get all the possible message ids for n

provebyinstantiation
                using: allowed_message_ids
                substitutions: (x=n),

; Run the cases for the message identifiers mentioned in the
; comment above, and defer all other cases

mcases
   (case: msg_id(n) = 61
     proof:
           (setflag traceflag on,
            comment \"Currently handling message id 61...\",
            setflag traceflag off,

            interpret every_case.proof,

            setflag traceflag on,
            comment \"Finished with message id 61....\",
            setflag traceflag off)

    case: msg_id(n) = 62
     proof:
           (setflag traceflag on,
            comment \"Currently handling message id 62...\",
            setflag traceflag off,

            interpret every_case.proof,

            setflag traceflag on,
            comment \"Finished with message id 62....\",
            setflag traceflag off)


    case: (msg_id(n) ~= 61 & msg_id(n) ~= 62)
     proof:
           (setflag traceflag on,
            comment \"Deferring proofs for all ids except : 61 and 62... \",
            setflag traceflag off,

            defer)),

interpret printdate_if_wanted,

close,

setflag traceflag on,
      comment \"Finished with interpretation of step_case.proof...\",
      setflag traceflag off)")

(defproof step_case.proof
    "(setflag traceflag on,
      comment \"Beginning interpretation of step_case.proof...\",
      comment \"This version of step_case.proof handles message ids: 63\",
      setflag traceflag off,

      interpret printdate_if_wanted,


      ; Get the possible message identifiers for n

      provebyinstantiation
                     using: allowed_message_ids
                     substitutions: (x=n),

      ; Run the cases for the message identifiers mentioned in the
      ; comment above, and defer all other cases

      mcases
         (case: msg_id(n) = 63
           proof:
                  (setflag traceflag on,
                   comment \"Currently handling message id 63...\",
                   setflag traceflag off,

                   interpret every_case.proof,

                   setflag traceflag on,
                   comment \"Finished with message id 63....\",
                   setflag traceflag off)
          case: msg_id(n) ~= 63
           proof:
                  (setflag traceflag on,
                   comment \"Deferring proofs for all message ids except : 63 \",
                   setflag traceflag off,
                   defer)),

      interpret printdate_if_wanted,

      close,

      setflag traceflag on,
      comment \"Finished with interpretation of step_case.proof...\",
      setflag traceflag off)")

(defproof step_case.proof
    "(setflag traceflag on,
      comment \"Beginning interpretation of step_case.proof...\",
      comment \"This version of step_case.proof handles message ids:
                  64, 65, and 66\",
      setflag traceflag off,
interpret printdate_if_wanted,


; Get all the possible message ids for n

provebyinstantiation
                using: allowed_message_ids
                substitutions: (x=n),

; Run the cases for the message identifiers mentioned in the
; comment above, and defer all other cases

mcases
   (case: msg_id(n) = 64
     proof:
           (setflag traceflag on,
            comment \"Currently handling message id 64...\",
            setflag traceflag off,

            interpret every_case.proof,

            setflag traceflag on,
            comment \"Finished with message id 64....\",
            setflag traceflag off)

    case: msg_id(n) = 65
     proof:
           (setflag traceflag on,
            comment \"Currently handling message id 65...\",
            setflag traceflag off,

            interpret every_case.proof,

            setflag traceflag on,
            comment \"Finished with message id 65....\",
            setflag traceflag off)

    case: msg_id(n) = 66
     proof:
           (setflag traceflag on,
            comment \"Currently handling message id 66...\",
            setflag traceflag off,

            interpret every_case.proof,

            setflag traceflag on,
            comment \"Finished with message id 66....\",
            setflag traceflag off)

    case: (msg_id(n) ~= 64 & msg_id(n) ~= 65 & msg_id(n) ~= 66)
     proof:
           (setflag traceflag on,
            comment \"Deferring proofs for all ids except :
                        64, 65, and 66 ... \",
            setflag traceflag off,
                    defer)),

        interpret printdate_if_wanted,

        close,

        setflag traceflag on,
        comment \"Finished with interpretation of step_case.proof...\",
        setflag traceflag off)")

(defproof step_case.proof
      "(setflag traceflag on,
        comment \"Beginning interpretation of step_case.proof...\",
        comment \"This version of step_case.proof handles message ids: 67\",
        setflag traceflag off,

        interpret printdate_if_wanted,


        ; Get the possible message identifiers for n

        provebyinstantiation
                       using: allowed_message_ids
                       substitutions: (x=n),

        ; Run the cases for the message identifiers mentioned in the
        ; comment above, and defer all other cases

        mcases
           (case: msg_id(n) = 67
             proof:
                   (setflag traceflag on,
                    comment \"Currently handling message id 67...\",
                    setflag traceflag off,

                    interpret every_case.proof,

                    setflag traceflag on,
                    comment \"Finished with message id 67....\",
                    setflag traceflag off)
            case: msg_id(n) ~= 67
             proof:
                   (setflag traceflag on,
                    comment \"Deferring proofs for all message ids except : 67 \",
                    setflag traceflag off,
                    defer)),

        interpret printdate_if_wanted,

        close,

        setflag traceflag on,
        comment \"Finished with interpretation of step_case.proof...\",
        setflag traceflag off)")
```

```
(defproof step_case.proof
    "(setflag traceflag on,
      comment \"Beginning interpretation of step_case.proof...\",
      comment \"This version of step_case.proof handles message ids:
                  80, 81, 82, and 83\",
      setflag traceflag off,

      interpret printdate_if_wanted,


      ; Get all the possible message ids for n

      provebyinstantiation
                     using: allowed_message_ids
                     substitutions: (x=n),

      ; Run the cases for the message identifiers mentioned in the
      ; comment above, and defer all other cases

      mcases
         (case: msg_id(n) = 80
           proof:
                  (setflag traceflag on,
                   comment \"Currently handling message id 80...\",
                   setflag traceflag off,

                   interpret every_case.proof,

                   setflag traceflag on,
                   comment \"Finished with message id 80....\",
                   setflag traceflag off)

          case: msg_id(n) = 81
           proof:
                  (setflag traceflag on,
                   comment \"Currently handling message id 81...\",
                   setflag traceflag off,

                   interpret every_case.proof,

                   setflag traceflag on,
                   comment \"Finished with message id 81....\",
                   setflag traceflag off)

          case: msg_id(n) = 82
           proof:
                  (setflag traceflag on,
                   comment \"Currently handling message id 82...\",
                   setflag traceflag off,

                   interpret every_case.proof,

                   setflag traceflag on,
                   comment \"Finished with message id 82....\",
                setflag traceflag off)

        case: msg_id(n) = 83
         proof:
               (setflag traceflag on,
                comment \"Currently handling message id 83...\",
                setflag traceflag off,

                interpret every_case.proof,

                setflag traceflag on,
                comment \"Finished with message id 83....\",
                setflag traceflag off)

        case: (msg_id(n) ~= 80 & msg_id(n) ~= 81 & msg_id(n) ~= 82
               & msg_id(n) ~= 83)
         proof:
               (setflag traceflag on,
                comment \"Deferring proofs for all ids except :
                            80, 81, 82, and 83 ... \",
                setflag traceflag off,

                defer)),

    interpret printdate_if_wanted,

    close,

    setflag traceflag on,
    comment \"Finished with interpretation of step_case.proof...\",
    setflag traceflag off)")

(defproof step_case.proof
   "(setflag traceflag on,
    comment \"Beginning interpretation of step_case.proof...\",
    comment \"This version of step_case.proof handles message ids:
              84, 85, 86, and 87\",
    setflag traceflag off,

    interpret printdate_if_wanted,


    ; Get all the possible message ids for n

    provebyinstantiation
                  using: allowed_message_ids
                  substitutions: (x=n),

    ; Run the cases for the message identifiers mentioned in the
    ; comment above, and defer all other cases

    mcases
       (case: msg_id(n) = 84
         proof:
               (setflag traceflag on,
         comment \"Currently handling message id 84...\",
         setflag traceflag off,

         interpret every_case.proof,

         setflag traceflag on,
         comment \"Finished with message id 84....\",
         setflag traceflag off)

case: msg_id(n) = 85
 proof:
        (setflag traceflag on,
         comment \"Currently handling message id 85...\",
         setflag traceflag off,

         interpret every_case.proof,

         setflag traceflag on,
         comment \"Finished with message id 85....\",
         setflag traceflag off)

case: msg_id(n) = 86
 proof:
        (setflag traceflag on,
         comment \"Currently handling message id 86...\",
         setflag traceflag off,

         interpret every_case.proof,

         setflag traceflag on,
         comment \"Finished with message id 86....\",
         setflag traceflag off)

case: msg_id(n) = 87
 proof:
        (setflag traceflag on,
         comment \"Currently handling message id 87...\",
         setflag traceflag off,

         interpret every_case.proof,

         setflag traceflag on,
         comment \"Finished with message id 87....\",
         setflag traceflag off)

case: (msg_id(n) ~= 84 & msg_id(n) ~= 85 & msg_id(n) ~= 86
        & msg_id(n) ~= 87)
 proof:
        (setflag traceflag on,
         comment \"Deferring proofs for all ids except :
                     84, 85, 86, and 87 ... \",
         setflag traceflag off,

         defer)),
      interpret printdate_if_wanted,

      close,

      setflag traceflag on,
      comment \"Finished with interpretation of step_case.proof...\",
      setflag traceflag off)")

(defproof step_case.proof
    "(setflag traceflag on,
      comment \"Beginning interpretation of step_case.proof...\",
      comment \"This version of step_case.proof handles message ids:
                  88, 89, and 90\",
      setflag traceflag off,

      interpret printdate_if_wanted,


      ; Get all the possible message ids for n

      provebyinstantiation
                     using: allowed_message_ids
                     substitutions: (x=n),

      ; Run the cases for the message identifiers mentioned in the
      ; comment above, and defer all other cases

      mcases
         (case: msg_id(n) = 88
          proof:
                (setflag traceflag on,
                 comment \"Currently handling message id 88...\",
                 setflag traceflag off,

                 interpret every_case.proof,

                 setflag traceflag on,
                 comment \"Finished with message id 88....\",
                 setflag traceflag off)

          case: msg_id(n) = 89
          proof:
                (setflag traceflag on,
                 comment \"Currently handling message id 89...\",
                 setflag traceflag off,

                 interpret every_case.proof,

                 setflag traceflag on,
                 comment \"Finished with message id 89....\",
                 setflag traceflag off)

          case: msg_id(n) = 90
          proof:
                (setflag traceflag on,
                 comment \"Currently handling message id 90...\",
                 setflag traceflag off,

                 interpret every_case.proof,

                 setflag traceflag on,
                 comment \"Finished with message id 90....\",
                 setflag traceflag off)

         case: (msg_id(n) ~= 88 & msg_id(n) ~= 89 & msg_id(n) ~= 90)
          proof:
                (setflag traceflag on,
                 comment \"Deferring proofs for all ids except :
                             88, 89, and 90 ... \",
                 setflag traceflag off,

                 defer)),

     interpret printdate_if_wanted,

     close,

     setflag traceflag on,
     comment \"Finished with interpretation of step_case.proof...\",
     setflag traceflag off)")

(defproof step_case.proof
   "(setflag traceflag on,
     comment \"Beginning interpretation of step_case.proof...\",
     comment \"This version of step_case.proof handles message ids: 91\",
     setflag traceflag off,

     interpret printdate_if_wanted,


     ; Get the possible message identifiers for n

     provebyinstantiation
                using: allowed_message_ids
                substitutions: (x=n),

     ; Run the cases for the message identifiers mentioned in the
     ; comment above, and defer all other cases

     mcases
        (case: msg_id(n) = 91
          proof:
                (setflag traceflag on,
                 comment \"Currently handling message id 91...\",
                 setflag traceflag off,

                 interpret every_case.proof,

                 setflag traceflag on,
                 comment \"Finished with message id 91....\",
                setflag traceflag off)
        case: msg_id(n) ~= 91
         proof:
               (setflag traceflag on,
                comment \"Deferring proofs for all message ids except : 91 \",
                setflag traceflag off,
                defer)),

    interpret printdate_if_wanted,

    close,

    setflag traceflag on,
    comment \"Finished with interpretation of step_case.proof...\",
    setflag traceflag off)")

(defproof step_case.proof
   "(setflag traceflag on,
    comment \"Beginning interpretation of step_case.proof...\",
    comment \"This version of step_case.proof handles message ids: 92 and 93\",
    setflag traceflag off,

    interpret printdate_if_wanted,


    ; Get all the possible message ids for n

    provebyinstantiation
                   using: allowed_message_ids
                   substitutions: (x=n),

    ; Run the cases for the message identifiers mentioned in the
    ; comment above, and defer all other cases

    mcases
       (case: msg_id(n) = 92
         proof:
               (setflag traceflag on,
                comment \"Currently handling message id 92...\",
                setflag traceflag off,

                interpret every_case.proof,

                setflag traceflag on,
                comment \"Finished with message id 92....\",
                setflag traceflag off)

        case: msg_id(n) = 93
         proof:
               (setflag traceflag on,
                comment \"Currently handling message id 93...\",
                setflag traceflag off,

                interpret every_case.proof,
                setflag traceflag on,
                comment \"Finished with message id 93....\",
                setflag traceflag off)


        case: (msg_id(n) ~= 92 & msg_id(n) ~= 93)
         proof:
               (setflag traceflag on,
                comment \"Deferring proofs for all ids except : 92 and 93... \",
                setflag traceflag off,

                defer)),

    interpret printdate_if_wanted,

    close,

    setflag traceflag on,
    comment \"Finished with interpretation of step_case.proof...\",
    setflag traceflag off)")
```

9.2 The main proof

```
; -*- Syntax: Common-lisp; Package: USER; Mode: LISP -*-~%
; File name : /u/versys/msx/infinite_sequence_data_structure_messages.proof

;;; This file contains the main proof and the proof prelude.

;; Proof prelude does various tasks that are needed before the proof of
;; infinite_sequence_data_structure_messages.sd begins.

(defproof prelude
    "(setflag traceflag on,
      comment  \ "Entering prelude... \ ",
      setflag traceflag off,

      ; load the new adatr fixes...
      eval (load_adatr_fix),

      ; Load the lisp file with the new proof commands
      eval (load  \ "/u/versys/msx/new_proof_commands \ "),

      ; Translate the program.
      adatr  \ "/u/versys/msx/msx_program_final_version.a \ ",

      ; Read in the file containing definitions, lemmas, etc. needed in
      ; the main proof.
       read  \ "/u/versys/msx/infinite_sequence_data_structure_messages.defs \ ",

      setflag traceflag on,
      comment  \ "Exiting prelude... \ ",
      setflag traceflag off)")


;; Proof main.proof is the top-level proof of the correctness assertion for
;;the program.


(defproof main.proof
    "(interpret prelude,
      setflag autoclose off,

      prove infinite_sequence_data_structure_messages.sd
       proof:
         (setflag traceflag on,
          comment  \ "Starting main proof... \ ",
          setflag traceflag off,

          let timing = false,

          interpret printdate_if_wanted,

          ;Since the postcondition has the form x ge 1 --> ..., we
          ;immediately get rid of the trivial case ~(x ge 1).

          cases ~(x ge 1)
           then proof:
                       (setflag traceflag on,
                        comment  \ "Trivial case... \ ",
                        setflag traceflag off,
                        close)

           else proof:
            (setflag traceflag on,
             comment  \ "Starting non-trivial case... \ ",
             setflag traceflag off,

             interpret printdate_if_wanted,

             ; Go to the beginning of the main loop in the
             ; program, handling the constant arrays

             interpret fix_constant_arrays,
             interpret go_to_beginning_of_loop,
             interpret do_lets,

             comment  \ "Starting induction... \ ",

             induct on:     n
               from:        1
               to:          x + 1
               invariants:
                 (
                  ;Needed for covering information

                  pcovering(.msx_program_final_version,old_universe),
   ;Variable values

   .build_in_progress = false,
   .cmd_count = 0,
   .lost_cmd = false,
   .app_msg_counter = 0,

   ;Queue variables and bounds

   .q_head
      = 1 + .q_tail mod .app_msg_q_size,
   .q_head ge origin(app_msg_q),
   .q_head
      le (origin(app_msg_q) +
          range(app_msg_q)) - 1,
   .q_tail mod .app_msg_q_size + 1
      ge origin(app_msg_q),
   .q_tail mod .app_msg_q_size + 1
      le (origin(app_msg_q) +
          range(app_msg_q)) - 1,

   ;State delta at beginning of loop

   formula(loopsd),

   ;Input/output counters are where they should be

   msg_input_begins_at(n) = .stdin \\ ctr,
   msg_output_begins_at(n) = .stdout \\ ctr,

   ;Output condition is true after execution of loop when
   ;n = x. This is stated in this way due to efficiency
   ;considerations (i.e., the more straightforward way
   ;introduces many new places - see report for more details).

   n = x + 1 --> formula(output_condition))

comodlist:
   (diff
     (old_universe,
        union(
          cmd_buf,cmd_status_buf,cmd_status,
          lost_cmd,cmd_count,fifo_not_empty_count,
          io_status_word,app_msgs.cmd_count,
          app_msgs.cmd_bstatus_buf,
          app_msgs.lost_cmd,
          app_msgs.fifo_not_empty_count,
          rejected_cmd_comsg,app_msg_q,
          build_in_progress,scmd_with_par,
          word_cntr,cmds_rcmds_for_app_msg,
          num_words_for_app_msg,contail,q_head,
          app_msg_counter,cmd_last_app_msg,
          saved_sat_subsys_config,load_msg,cmd_total,
          app_msgs.io_status_word,stdout \\ ctr,
             msx_program_final_version \\ pc,
             msx_program_final_version,stdin \\ ctr,stdout)))

   modlist:
      (cmd_buf,cmd_status_buf,cmd_status,lost_cmd,
       cmd_count,fifo_not_empty_count,
       io_status_word,app_msgs.cmd_count,
       app_msgs.cmd_buf,app_msgs.cmd_status_buf,
       app_msgs.lost_cmd,
       app_msgs.fifo_not_empty_count,
       rejected_cmd_count,app_msg,app_msg_q,
       build_in_progress,starting_cmd_with_par,
       word_cntr,cmds_rcvd,num_cmds_for_app_msg,
       num_words_for_app_msg,cont_cmd,q_tail,q_head,
       app_msg_counter,cmd_last_app_msg,
       saved_sat_subsys_config,load_msg,cmd_total,
       app_msgs.io_status_word,stdout \\ ctr,
       msx_program_final_version \\ pc,
       msx_program_final_version,stdin \\ ctr,stdout,
       diff(all,old_universe))

    base proof:  interpret base_case.proof

    step proof:  interpret step_case.proof,

  interpret printdate_if_wanted,

  setflag traceflag on,
  comment  \ "Finishing off induction .. \ ",
  apply finale.sd,

  setflag traceflag off,
  close),

 interpret printdate_if_wanted,

 setflag traceflag on,
 comment  \ "Finishing off main proof... \ ",
 setflag traceflag off,

 close),

setflag traceflag on,
comment  \ "Main proof completed \ ",
interpret printdate_if_wanted)")
```

References

[1] T. K. Menas, "A Proposal for the Verification in SDVS of a Portion of the MSX Tracking Processor Software," Technical Report ATR-92(2778)-7, The Aerospace Corporation, September 1992.

[2] G. Heyler, S. Hutton, and R. L. Waddell, "Midcourse Space Experiment (MSX) Tracking Processor Software Detailed Design Document," Technical Report S1A-084-91, JHU/APL, 1991.

[3] R. L. Waddell and S. Hutton, "Midcourse Space Experiment (MSX) Tracking Processor Software Functional Design," Technical Report S1A-031-90, JHU/APL, 1990.

[4] S. Hutton, "Tracking Processor / Command Processor Interface Design Specification (Version 2)," Technical Report S1A-136-91, JHU/APL, 1991.

[5] U. S. Department of Defense, *Reference Manual for the Ada Programming Language (ANSI/MIL-STD-1815A)*, 22 January 1983.

[6] T. K. Menas, J. M. Bouler, J. E. Doner, I. F. Filippenko, B. H. Levy, and L. G. Marcus, "Overview of the MSX Verification Experiment using SDVS," Technical Report ATR-93(3778)-6, The Aerospace Corporation, September 1993.